UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Final de Grado

Realización de un solver automático de problemas de programación lineal

Autor: Carlos Clavero Muñoz

Junio, 2016

Tutor: Carlos Linares López

Agradecimientos

A mis padres José y Begoña, porque sin ellos nada sería posible. Gracias por apoyarme, aconsejarme y sacrificarse para que pase lo que pase, nunca deje de avanzar.

A mi abuela Mani, porque sé que le hace inmensamente feliz que termine este trabajo y esta etapa.

A mi hermano Dani, por ayudarme a trabajar siempre de la mejor manera y por buscar la forma de que mi trabajo sea siempre lo prioritario.

A mis amigos, por ser mi otra familia y estar siempre ahí para impulsarme cuando lo necesito.

A mi novia Blanca, por escucharme, animarme y ser mi apoyo y mi alegría, en todo momento. Y sobre todo, por ayudarme a creer en mí mismo.

A mis compañeros de prácticas, Pablo y David, por todos los momentos difíciles que nos ha tocado superar durante la carrera.

A mi tutor Carlos Linares, por su dedicación y por darme la posibilidad de trabajar con él y aprender muchísimo.

Y, por último, como todas las cosas importantes de mi vida, quiero dedicar este trabajo a mi abuelo Jesús, porque sin sus consejos, sus enseñanzas y nuestras infinitas conversaciones, nunca habría llegado hasta aquí. Allá donde esté, este trabajo también es suyo.

Índice

Índice de tablas

Índice de ilustraciones

Capítulo 1
Introducción

1. Introducción

La programación lineal, es un campo de optimización matemática en el que se busca la optimización (maximización o minimización) de una función lineal a la que llamaremos función objetivo, de tal forma que las variables que en esta función aparecen, estarán sujetas a un conjunto de restricciones, que serán también lineales. Este tipo de problemas, como más adelante se verá, pueden ser resueltos de varias maneras aunque en este proyecto nos centraremos en el algoritmo *Simplex* y en la resolución gráfica. [1]

Leyendo su definición, podría parecer que la programación lineal trata problemas muy complejos, pero realmente, no tiene porqué. Se pueden modelar problemas de programación lineal para problemas tan cotidianos como pueda ser, qué autobús realiza qué recorrido en función del gasto que realiza cada uno.

El método *Simplex* nos permite resolver este tipo de problemas de manera metódica y por lo tanto sencilla. Es decir, se podrá alcanzar la solución de cualquier problema mediante la utilización de este método, si ésta existe, y en tal caso de que no existiera también nos permitirá llegar a esa conclusión.

Por supuesto, al tratarse este método de la aplicación de pasos durante varias iteraciones, el tiempo para alcanzar la solución aumentará exponencialmente cuando aumente la complejidad. La complejidad puede aumentar de varias formas: aumentando el número de variables, aumentando el número de restricciones, trabajando con números más grandes, etc.

El problema que presenta la utilización del método *Simplex*, es el gran número de cálculos matemáticos que hay que aplicar para la resolución, algo que resulta tedioso, mucho más cuando el problema se alarga en cuanto a iteraciones se refiere, o cuando el número de variables, lo hace muy costoso de resolver a mano.

Bastante más simple y rápida, es la aplicación del método gráfico, que también permite la resolución de este tipo de problemas. Sin embargo, para tener garantías de una correcta resolución mediante este método, el problema solo puede tener dos variables, ya que con tres es complicado de dibujar y con más de tres resulta imposible de representar. Aun así, es complicado que una persona mediante una representación gráfica a mano, obtenga los puntos exactos que pueden ser solución del problema.

Por lo tanto, se use el método que se use, aparecen inconvenientes. Estas desventajas, son las que este proyecto final de grado pretende eliminar o al menos minimizar. La idea es tener por un lado, un *software* que sea capaz de aplicar el método *Simplex* de forma automática, de tal manera, que los largos y tediosos cálculos sean realizados por una máquina, consiguiendo de esta manera reducir el tiempo y además, conseguir evitar los errores que una persona podría realizar al resolver cálculos matemáticos a mano. Por otro lado, y complementariamente, el mencionado *software* debería ser capaz de representar restricciones, puntos y regiones con la exactitud que una máquina puede ofrecer, terminando así con los problemas del método gráfico.

Este proyecto nace en el ámbito docente. El principal objetivo, es que cuando alumnos de cualquier campo estén aprendiendo a resolver problemas de programación lineal, puedan usar el *software* desarrollado en este proyecto final de grado para practicar, ver dónde fallan, conocer la solución de cualquier problema que se les ocurra, etc.

Además, se pretende que el uso de este proyecto, no se limite al método *Simplex*, ni siquiera al campo de la programación lineal, sino que cualquiera que lo desee tenga acceso a cualquier parte del proyecto para completar sus propios desarrollos, siempre que estos sean en el lenguaje *Python*, que será el lenguaje de desarrollo del sistema.

Capítulo 2
Estado de la cuestión

2. Estado de la cuestión

En este capítulo, se va a tratar cuál es la situación y el entorno actual que rodean al proyecto. También se incluirán estudios y explicaciones de algunos aspectos básicos, para entender el mismo.

2.2. Historia de la programación lineal

Durante los siglos XVII y XVIII, importantes y famosos matemáticos de la época, como Newton (1643-1727), Leibnitz (1646-1716), Bernoulli (1667–1748) y, sobre todo, Lagrange (1736-1813), los cuales habían contribuido fervientemente al desarrollo del cálculo infinitesimal, se dedican a la búsqueda de máximos y mínimos condicionados de determinadas funciones. [2]

Más adelante, sería el francés Jean Baptiste-Joseph Fourier (1768-1830), el primero en intuir los métodos de lo que hoy conocemos como programación lineal. Su aproximación, aunque imprecisa, se centró en la resolución de sistemas de inecuaciones lineales (aquí surge el método de eliminación de Fourier-Motzkin). [1]

Sin embargo, y exceptuando a Gaspar Monge (1746-1818), quien en 1776 realizó algunos estudios en este campo, no será hasta 1939, cuando se vuelvan a producir avances en programación lineal. Fue en ese año cuando el matemático ruso Leonid Vitalevich Kantorovitch (1912-1986), hizo una publicación llamada "Métodos matemáticos de organización y planificación de la producción", en la que mostraba un amplio conjunto de problemas que se correspondía con una teoría matemática precisa y bien definida. Esta teoría, se convertirá en lo que hoy llamamos programación lineal. Estas publicaciones realizadas en el campo de la programación lineal y más centrada en la economía, se anticiparon a las realizadas posteriormente por G. B. Dantzig (1914-2005). Gracias a esta y otras publicaciones, Kantorovitch ganó en 1975 el premio Nobel de economía.

En 1941, se formula por primera vez el problema del transporte, por Kantorovitch y Koopmans (1910-1985). Este problema es uno de los problemas más conocidos de la programación lineal. El problema del transporte consiste en calcular el coste de abastecer una serie de puntos de demanda a partir de un grupo de puntos de oferta, teniendo en cuenta los distintos precios de envío de cada punto de oferta a cada punto de demanda.

Será en los años posteriores a la Segunda Guerra Mundial, cuando la cuestión se relance de nuevo. Estados Unidos necesitaba de los modelos de optimización de la programación lineal, para resolver sus problemas de asignación de energías y recursos, puesto que eran altamente complejos. Ya durante la guerra, se habían usado estos métodos para planificar y mejorar los costes del ejército.

Paralelamente se empezaban a desarrollar las técnicas de computación y los ordenadores, que permitirían la simplificación de los problemas descritos.

En 1947, sería Dantzig quien describiría de forma exacta y precisa el enunciado al que debe quedar reducido todo problema de programación lineal. Dantzig junto con

otros miembros de la United States Departament of Air Force formarían el llamado grupo SCOOP (Scientific Computation of Optimum Programs).

En ese mismo año, Dantzig publicó el algoritmo *Simplex* y Von Neumann (1903-1957), contribuyó con la teoría de la dualidad.

Posteriormente, en 1979, el ruso Leonid Khachiyan (1952-2005) demostró mediante su Algoritmo del Elipsoide, que el problema de programación lineal es resoluble de forma eficiente y en un tiempo polinominal. Por último, en 1984, Narendra Karmarkar constituye un enorme avance en el área, mediante la inclusión de un nuevo método del punto interior para resolver problemas de programación lineal.

Para comprender la utilidad de la programación lineal, podemos utilizar un problema formulado por Dantzig. En él, se pretende asignar 70 personas a 70 puestos de trabajo. Si intentamos ver todas las combinaciones llegaremos a un número que sería 70!, lo cuál excede del número de partículas del universo. Sin embargo aplicando un enfoque de programación lineal, junto con el método *Simplex*, llegamos rápidamente a la solución óptima puesto que se descartan un gran número de posibles soluciones factibles. [2]

2.3. Método Simplex

El método *Simplex*, es un método consistente en una serie de pasos iterativos, a aplicar sobre un problema de programación lineal, y que nos permite obtener la solución del mismo. Como se ha podido ver antes, el método *Simplex* fue desarrollado por George Dantzing en 1947, y consiste en definir un poliedro como una región factible, a partir de un sistema de desigualdades lineales. El algoritmo *Simplex* comienza en un vértice y se mueve a lo largo de las aristas del poliedro, hasta alcanzar el vértice que proporciona la solución óptima. A continuación se va a ver, paso a paso, como se aplica el método *Simplex*.

2.3.1. Entrada al método Simplex

En primer lugar, el método *Simplex* va a recibir un problema que debe seguir una serie de condiciones. El problema de programación lineal deberá ser:

$$Optimizar\ z = c^T x$$
$$sujeto\ a\ \ Ax \leq b\ , x \geq 0$$

Donde z será la función objetivo, es decir, la función que hay que optimizar; c^T será el vector de coeficientes de las variables de decisión, para la función objetivo; A será la matriz de coeficientes, que representará los coeficientes de las variables de decisión en cada restricción; y por último, la b será el vector de recursos. Además debe cumplirse que todas las variables de decisión tomen un valor positivo o 0.

2.3.2. Forma normal estándar

El primer paso, a la hora de resolver un problema mediante el algoritmo *Simplex*, es transformar el problema a forma estándar.

Se dice que un problema de programación lineal, está en forma normal estándar, si y solo si:

- El objetivo de la función es maximizar.
- Todas las restricciones, son del tipo =.
- Todas las variables de decisión son ≥ 0.
- Todos los recursos son no negativos, es decir, que ningún elemento del vector de recursos puede tomar un valor negativo.

Por supuesto, cuando nos enfrentamos a un problema de programación lineal, al modelarlo no se nos suele presentar un problema en forma estándar, luego se deben hacer las transformaciones pertinentes para que esto ocurra. Las transformaciones son las siguientes:

- Transformar la función de minimización en una función de maximización:

$$\min c^T \mathrm{x} \; = \; - \max c^T \mathrm{x}$$

Ejemplo:

$$\min 3x_1 + x_2 - 4x_3 \; = \; - \max - 3x_1 - x_2 + 4x_3$$

- Transformar restricciones de la forma $\leq$ o $\geq$ a la forma =:

$$\sum_{j=1}^{n} a_j + x_j \leq b_i \; = \sum_{j=1}^{n} a_j + x_j + s_i = b_i$$

$$\sum_{j=1}^{n} a_j + x_j \geq b_i \; = \sum_{j=1}^{n} a_j + x_j - t_i = b_i$$

Estas variables que se añaden a las restricciones, se denominan variables de holgura y se añaden también a la función objetivo, pero con coeficiente 0.

Ejemplo:

$$3x_1 + 2x_2 + 4x_3 \leq 2 \; = \; 3x_1 + 2x_2 + 4x_3 + x_4 = 2$$

- Transformar los recursos a positivo

$$\sum_{j=1}^{n} a_j + x_j \leq b_i \; = - \sum_{j=1}^{n} a_j + x_j \geq -b_i$$

Ejemplo:

$$3x_1 + 2x_2 + 4x_3 \leq -2 \; = \; -3x_1 - 2x_2 - 4x_3 \geq 2$$

Las anteriores transformaciones, son los pasos básicos para llegar a la forma estándar. Sin embargo, aún se necesita hacer una última transformación para poder

comenzar a aplicar el algoritmo *Simplex*. La transformación consiste en añadir las denominadas variables artificiales cuando sea necesario.

Cuando se presente una restricción de tipo $\geq$, como se ha visto más arriba, es necesario añadir una variable de holgura para transformarla a la forma $=$. Esta variable se añade con signo negativo. Pues bien, es necesario añadir otra variable más a la restricción, ahora con signo positivo. Más adelante, en el apartado 2.3.3, se verá el motivo de añadir esta variable y las consecuencias que puede generar. En conclusión, la transformación completa de una restricción de tipo $\geq$, sería así:

$$\sum_{j=1}^{n} a_j + x_j \geq b_i \; = \sum_{j=1}^{n} a_j + x_j - t_i + s_i = b_i$$

Las variables artificiales se añaden a la función objetivo con un coeficiente de - ∞.

Una vez realizadas todas estas transformaciones, se puede empezar a aplicar el método *Simplex*.

2.3.3. Selección de una base factible inicial

En este paso, se selecciona una base. La base será una matriz cuadrada, tomando los coeficientes de las variables de decisión, holgura y artificiales, presentes en las restricciones. Es decir las columnas de la matriz A de coeficientes. Para la primera iteración, se coge como base la matriz identidad. Este es el motivo por el cuál hay que añadir las variables artificiales, para forzar la aparición de la matriz identidad, cuando haya restricciones $\leq$. Por lo tanto, si se diera el caso de que ya hay un coeficiente en esa restricción que permite sacar la matriz identidad, no sería necesario añadir ninguna variable artificial.

2.3.4. Cálculo de la solución de la iteración

A continuación se calcula la solución de la iteración, es decir, qué valor toman las variables seleccionadas en esta iteración. Para ello:

$$x_0 = B_0^{-1} b$$

Donde x_0 es la solución de la primera iteración, B_0 es la base de la primera iteración y b , es el vector de recursos.

2.3.5. Calculo del valor de la función de la iteración

Ahora con los valores de las variables obtenidos en la solución de la iteración se calcula el valor de la función para la primera iteración. Para ello:

$$z_0 = c_{B_0}^T x_0$$

Donde z_0 es el valor de la función para la primera iteración, $c_{B_0}^T$ es el valor del vector de la función para la iteración (es decir, cogemos solo los coeficientes del vector de la función de aquellas variables que pertenecen a la iteración), y x_0 es la solución de la iteración.

2.3.6. Regla de entrada

A continuación, calculamos los valores de la regla de entrada, que nos permitirá saber qué variable será la que entre en la siguiente iteración del problema. Para ello, calculamos lo siguiente, para todas aquellas variables que no pertenecen a la solución:

$$z_j - c_j = c_{B_0}^T B_0^{-1} a_j - c_j$$

Normalmente, $B_0^{-1} a_j$ se calcula previamente para todas las variables que no pertenecen a la iteración, y se le da el nombre de y_j.

La variable que de un valor más negativo de $z_j - c_j$, será la variable que entrará en la siguiente iteración. Si dos variables presentan el mismo valor, es indiferente, coger cualquiera de las dos.

2.3.7. Regla de salida

Se calculan ahora los valores de la regla de salida. Estos valores nos marcarán cuál será la variable que salga en la iteración. Para ello, se calcula:

$$O = min_{i \in B}\{\frac{x_i}{y_{ii'}}\}$$

Donde O será el valor mínimo resultante de la regla de salida, x_i será la solución de la iteración, e $y_{ii'}$ será el valor de y de la variable que entra en la siguiente iteración. Recordemos que $y_j = B_0^{-1} a_j$, como vimos en el punto anterior.

Es importante resaltar dos cosas. La primera, es que se descartarán de la ecuación anterior todos aquellos valores que presenten un denominador negativo o igual a 0. La segunda, es que la variable que saldrá será aquella que proporcione el valor más pequeño de O, es decir, la que presente el mínimo x/y.

2.3.8. Mejora de la solución

Lo que se hace en este paso, es calcular la base para la nueva iteración. Para ello, se cogen las variables de la base anterior, se saca la variable que se calculó que saldría en la variable de salida, y se mete la variable que se calculó que entraría en la regla de entrada. Por ejemplo:

1. Si la base de la iteración anterior está formada por las variables $[x_3, x_4, x_5]$.
2. Se calcula en la regla de salida que sale la variable x_4.
3. Se calcula en la regla de entrada que entra la variable x_1.
4. La base de la nueva iteración va a ser la formada por las variables $[x_1, x_3, x_5]$.

Una vez calculada la base de la iteración, se procede a realizar los mismos pasos que se realizaron anteriormente, es decir hay que volver al punto 2.3.4.

2.3.9. Finalización del problema y posibles soluciones

La dinámica a seguir, es por tanto, la vista en los pasos anteriores, mejorando continuamente la solución. Se va a tratar en este punto, cuál es el momento en el que se deja de buscar una mejora de la solución actual, es decir cuando el problema finaliza. Se pueden dar varios casos, indicando diferentes formas de solución:

- **Cuando la variable de entrada no presenta ningún valor negativo.** En teoría, para saber qué variable entrará en la siguiente iteración, hay que seleccionar la que tenga un valor más negativo en la regla de entrada. Pues bien, si no existe ningún valor negativo, querrá decir que no hay ninguna variable cuya entrada en la base mejore la solución actual, luego el problema ha terminado. En este caso, el problema tendrá solución única y será la de la iteración actual.
- **Cuando la variable de entrada presenta un valor 0 y el resto de costes positivos.** Como ocurría en el caso anterior, si al calcular la variable que entrará en la siguiente iteración resulta que no hay costes negativos y además alguna variable presenta coste 0, querrá decir que esa variable al entrar, produce el mismo valor en la función objetivo (función a optimizar), así que el problema ha finalizado. Se dice en este caso, que el problema tiene un número infinito de soluciones.
- **Cuando no podemos sacar ninguna variable de la iteración.** Podría darse el caso de que sí que haya costes negativos en la variable de entrada, pero que al calcular los valores de la regla de salida, se observe que todos los denominadores son negativos o 0. En este caso el problema ha terminado. Decimos que el problema es no acotado, puesto que se puede mejorar el valor de la función objetivo de manera indefinida.
- **Cuando una variable artificial toma un valor positivo.** Como se vio más arriba, las variables artificiales se introducen en la función objetivo, con un coeficiente de - ∞. Si una variable artificial toma un valor positivo el valor de la función objetivo se tornará - ∞, algo que no tiene sentido. Luego, en caso de que alguna variable artificial tome valor positivo, se dirá que el problema es infactible.

2.3.10. Dualidad

Cuando se resuelve un problema de programación lineal, se dice que se ha resuelto el problema primal, ya que tiene una relación directa con la necesidad del planteamiento. Automáticamente, al planteamiento del problema primal, se puede plantear otro problema que puede ser resuelto y que tiene importantes relaciones y propiedades con respecto al problema primal. Este problema, se denomina problema dual, y más adelante, en el apartado 2.3.11, al ver la interpretación de las soluciones, se verá el significado que el mismo puede aportar a la solución del problema primal.

2.3.11. Transformación problema primal-problema dual

Para transformar un problema primal en uno dual, habrá que tener el problema primal, en forma simétrica de maximización.

Se dice que un problema de programación lineal está en forma simétrica de maximización, si y solo si:

- La función objetivo, es de maximización.
- Todas las restricciones son <=.
- Todas las variables de decisión toman valores positivos o nulos.

Además de las transformaciones que se hacían para alcanzar la forma estándar (consultar punto 2.3.2), las cuáles son perfectamente aplicables, en caso de que alguna facilite la obtención de la forma simétrica de maximización, hay que añadir la siguiente transformación:

$\sum_{j=1}^{n} a_j + x_j = b_i$ Será igual a las dos desigualdades siguientes:

$$\sum_{j=1}^{n} a_j + x_j \leq b_i$$

$$-\sum_{j=1}^{n} a_j + x_j \leq -b_i$$

Una vez aplicadas las transformaciones pertinentes, se obtiene un problema en la forma simétrica de maximización. A continuación se puede ver la transformación del problema primal (en forma simétrica) al problema dual:

Problema primal		Problema dual
$\max Z = c^T x$		$\min w = b^T x'$
$Ax \leq b$	⇨	$A^T x' \leq c$
$x \geq 0$		$x' \geq 0$

2.3.12. Obtención de la solución del problema dual

Si el problema de programación lineal en forma simétrica tiene una solución óptima correspondiente a una base B, entonces x' (solución del problema dual):

$$x' = c_B^T B^{-1}$$

Es decir, la solución del problema dual se obtiene al multiplicar el vector transpuesto de la función objetivo para la iteración (es decir, solo con los coeficientes de las variables de la iteración) con la que se obtuvo la solución óptima del problema dual, multiplicado por la inversa de la base de dicha iteración.

La solución del problema dual, representa la contribución por unidad de recurso i-ésimo, al crecimiento de la función objetivo. En el siguiente apartado, se puede ver un ejemplo más detallado de lo que esto quiere decir.

2.3.13. Interpretación de la solución

Una vez obtenida la solución de un problema de programación lineal, se puede afirmar lo siguiente sobre él:

- **Factible/Infactible**: es decir, si el problema tiene solución o no.
- **Solución única/Infinitas soluciones**: es decir, si es una única solución la que obtiene el máximo valor de la función objetivo, o por el contrario, son varias las soluciones que permiten alcanzar ese valor.
- **Acotado/No acotado**: es decir, si la función objetivo toma un único valor óptimo o si por el contrario la función objetivo mejora de manera indefinida.

Todas estas afirmaciones, son con respecto a la solución, pero además podemos hacer otras afirmaciones con respecto a los recursos:

- **Significado de los valores de holgura**. Si una variable de holgura, toma un valor de por ejemplo, 3, quiere decir que sobran 3 recursos. Si tomara un valor de -3, querría decir que faltan 3 recursos.
- **Contribución por unidad de recursos**. Este aspecto, se calcula mediante la solución del problema dual. Si por ejemplo, se obtiene una solución del problema dual (a partir de la última iteración del problema primal, que contaba con las variables x_1 y x_2) que es (2 1), quiere decir que por cada unidad que aumenta un recurso de x_1, la función objetivo aumenta en dos unidades; y por cada unidad de x_2, supone una unidad en la función objetivo.

2.4. Los números racionales

Los números *racionales* son aquellos que pueden expresarse como el cociente de dos números enteros, es decir como una fracción común $\frac{a}{b}$, siendo a y b distintos de 0. [3]

Por otro lado, los números *irracionales* son aquellos que no pueden ser expresados como un cociente de dos números enteros, diferentes de 0. [4]

Por último, el conjunto de los números reales incluye tanto los números *racionales* como los *irracionales*.

Pues bien, cuando se resuelve el método *Simplex*, se puede elegir, con qué tipo de números se desea resolver el problema, y como se verá a continuación es muy conveniente usar los *racionales*, en lugar de los reales.

Por supuesto, no es obligatorio aplicar el método *Simplex* con números *racionales*, de hecho en el apartado 2.6, se verán algunos ejemplos de *software* que permiten la resolución del mismo, con números reales (es decir, incluyendo también números *irracionales*). Por otro lado, sí es muy recomendable utilizar los números *racionales*, sobre todo, si se quiere llegar a una solución exacta, ya que en un método con una carga de cálculos matemáticos como es el *Simplex*, se perderá exactitud en cada operación si se utilizan números reales.

Un ejemplo claro de la diferencia de operaciones con números reales y *racionales*, podría ser el siguiente:

Operación con números *racionales*	Operación con números reales
$\frac{1}{7}$ x $7 = 1$	1/7=0,1428571…
	0,1428571 x 7 = 0,9999997

Como se puede ver, la misma supuesta operación, no ofrece el mismo resultado, ya que en el primer caso, un 7 en el numerador y otro en el denominador, se anulan; mientras que en el segundo, vemos que el resultado nos queda como un número decimal no exacto, ya que para realizar la multiplicación es imposible coger todos los decimales de la operación 1/7, porque estos son infinitos. El problema de esta pérdida de exactitud, es que se va acumulando y se van arrastrando esas pequeñas diferencias, que van aumentando conforme se va desarrollando el problema.

En conclusión, durante el presente proyecto se buscará la máxima exactitud en la solución de los problemas de programación lineal, por lo tanto se utilizarán los números *racionales* en todas las operaciones.

2.5. Ejemplo de aplicación del Método Simplex

A continuación se va a ver un ejemplo de problema de programación lineal, resuelto mediante el método *Simplex*. Se irán desarrollando los pasos vistos en los problemas anteriores, hasta alcanzar la solución final.

Dado el problema:

$$\begin{gathered}\min Z = -2x_1 - x_2 + 3x_3 - 5x_4 \\ x_1 + 2x_2 + 2x_3 + 4x_4 \leq 40 \\ -2x_1 + x_2 - x_3 - 2x_4 \geq -8 \\ 4x_1 - 2x_2 + x_3 - x_4 \leq 10 \\ x \geq 0\end{gathered}$$

1. Transformación a forma normal Estándar:

$$\begin{gathered}\max Z = 2x_1 + x_2 - 3x_3 + 5x_4 \\ \begin{aligned} x_1 + 2x_2 + 2x_3 + 4x_4 + x_5 &= 40 \\ 2x_1 - x_2 + x_3 + 2x_4 + x_6 &= 8 \\ 4x_1 - 2x_2 + x_3 - x_4 + x_7 &= 10 \end{aligned} \\ x \geq 0\end{gathered}$$

2. Cálculo de la solución:

   Paso 0. Calcula de una solución factible inicial.

   a. Cálculo de las variables básicas. La primera iteración se inicia con una base igual a la matriz identidad de dimensión 3 puesto que hay hasta tres restricciones. Por lo tanto, son variables básicas en el primer paso x_5, x_6, x_7

      - Base inicial: $B_0 = I_3$
      - Base inicial invertida: $B_0^{-1} = I_3$

- Solución de la iteración: $x_0^* = B_0^{-1}b = b = \begin{pmatrix} 40 \\ 8 \\ 10 \end{pmatrix}$
- Valor de la función para esta iteración: $z_0^* = c_{B_0}^T x_0^* = (0\ 0\ 0)\begin{pmatrix} 40 \\ 8 \\ 10 \end{pmatrix} = 0$
- Valores de y: $y_1 = B_0^{-1}a_1 = \ a_1 = \begin{pmatrix} 1 \\ 2 \\ 40 \end{pmatrix}$

$$y_2 = B_0^{-1}a_2 = \ a_2 = \begin{pmatrix} 2 \\ -1 \\ -2 \end{pmatrix}$$

$$y_3 = B_0^{-1}a_3 = \ a_3 = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

$$y_4 = B_0^{-1}a_4 = \ a_4 = \begin{pmatrix} 4 \\ 2 \\ -1 \end{pmatrix}$$

b. Selección de la variable de entrada:

$$z_1 - c_1 = c_{B_0}^T y_1 - c_1 = (0\ 0\ 0)\begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix} - 2 = -2$$

$$z_2 - c_2 = c_{B_0}^T y_2 - c_2 = (0\ 0\ 0)\begin{pmatrix} 2 \\ -1 \\ -2 \end{pmatrix} - 1 = -1$$

$$z_3 - c_3 = c_{B_0}^T y_3 - c_3 = (0\ 0\ 0)\begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} + 3 = +3$$

$$z_4 - c_4 = c_{B_0}^T y_4 - c_4 = (0\ 0\ 0)\begin{pmatrix} 4 \\ 2 \\ -1 \end{pmatrix} - 5 = -5$$

Por lo tanto, entra la variable con el valor más negativo, es decir x_4. Si en este punto, se encuentra con un coste 0, y el resto de valores positivos, el problema habría terminado con infinitas soluciones.

c. Selección de la variable de salida:

$$\min\{\frac{x_0^*}{y_4}\}$$

$$\min\{\frac{40}{4}, \frac{8}{2}, \frac{\cancel{10}}{\cancel{-1}}\}$$

Si se descartan los denominadores negativos, o nulos, se obtiene que sale la variable x_6. Si todos los denominadores fueran nulos o negativos, el problema se habría terminado y sería no acotado.

Paso 1. Mejora de la solución actual.

a. Cálculo de las variables básicas. En esta nueva iteración las variables básicas serán x_4, x_5, x_7.

- Base de la iteración: $B_1 = \begin{pmatrix} 4 & 1 & 0 \\ 2 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}$
- Base de la iteración invertida: $B_1^{-1} = \begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 1 & -2 & 0 \\ 0 & \frac{1}{2} & 1 \end{pmatrix}$
- Solución de la iteración: $x_1^* = B_1^{-1}b = \begin{pmatrix} 4 \\ 24 \\ 14 \end{pmatrix}$
- Valor de la función para esta iteración: $z_1^* = c_{B_1}^T x_1^* = (5\ 0\ 0)\begin{pmatrix} 4 \\ 24 \\ 14 \end{pmatrix} = 20$
- Valores de y: $y_1 = B_1^{-1}a_1 = \begin{pmatrix} 1 \\ -3 \\ 5 \end{pmatrix}$

$$y_2 = B_1^{-1}a_2 = \begin{pmatrix} -\frac{1}{2} \\ 4 \\ -\frac{5}{2} \end{pmatrix}$$

$$y_3 = B_1^{-1}a_3 = \begin{pmatrix} \frac{1}{2} \\ 0 \\ \frac{3}{2} \end{pmatrix}$$

$$y_6 = B_1^{-1}a_6 = \begin{pmatrix} \frac{1}{2} \\ -2 \\ \frac{1}{2} \end{pmatrix}$$

b. Selección de la variable de entrada:

$$z_1 - c_1 = c_{B_1}^T y_1 - c_1 = (0\ 0\ 0)\begin{pmatrix} 1 \\ -3 \\ 5 \end{pmatrix} - 2 = 3$$

$$z_2 - c_2 = c_{B_1}^T y_2 - c_2 = (0\ 0\ 0)\begin{pmatrix} -\frac{1}{2} \\ 4 \\ -\frac{5}{2} \end{pmatrix} - 1 = \frac{7}{2}$$

$$z_3 - c_3 = c_{B_1}^T y_3 - c_3 = (0\ 0\ 0)\begin{pmatrix} \frac{1}{2} \\ 0 \\ \frac{3}{2} \end{pmatrix} + 3 = \frac{11}{2}$$

$$z_6 - c_6 = c_{B_1}^T y_6 - c_6 = (0\ 0\ 0)\begin{pmatrix} \frac{1}{2} \\ -2 \\ \frac{1}{2} \end{pmatrix} - 0 = \frac{5}{2}$$

Por lo tanto, entra la variable con el valor más negativo, es decir x_2.

c. Selección de la variable de salida:

$$\min\{\frac{x_1^*}{y_2}\}$$

$$\min\{\frac{4}{-\frac{1}{2}}, \frac{24}{2}, \frac{14}{-\frac{3}{2}}\}$$

Si se descartan los denominadores negativos, o nulos, se obtiene que sale la variable x_5.

Paso 2.Mejora de la solución actual.

a. Cálculo de las variables básicas. En esta nueva iteración las variables básicas serán x_2, x_4 , x_7.

- Base de la iteración: $B_2 = \begin{pmatrix} 2 & 4 & 0 \\ -1 & 2 & 0 \\ -2 & -1 & 1 \end{pmatrix}$

- Base de la iteración invertida: $B_2^{-1} = \begin{pmatrix} \frac{1}{4} & -\frac{1}{2} & 0 \\ \frac{1}{8} & \frac{1}{4} & 0 \\ \frac{5}{8} & -\frac{3}{4} & 1 \end{pmatrix}$
- Solución de la iteración: $x_2^* = B_2^{-1}b = \begin{pmatrix} 6 \\ 7 \\ 29 \end{pmatrix}$
- Valor de la función para esta iteración: $z_2^* = c_{B_2}^T x_2^* = (1\ 5\ 0)\begin{pmatrix} 6 \\ 7 \\ 29 \end{pmatrix} = 41$
- Valores de y: $y_1 = B_2^{-1}a_1 = \begin{pmatrix} -\frac{3}{4} \\ \frac{5}{8} \\ \frac{25}{8} \end{pmatrix}$

$$y_3 = B_2^{-1}a_3 = \begin{pmatrix} 0 \\ \frac{1}{2} \\ \frac{3}{2} \end{pmatrix}$$

$$y_5 = B_2^{-1}a_5 = \begin{pmatrix} \frac{1}{4} \\ \frac{1}{8} \\ \frac{5}{8} \end{pmatrix}$$

$$y_6 = B_2^{-1}a_6 = \begin{pmatrix} -\frac{1}{2} \\ \frac{1}{4} \\ -\frac{3}{4} \end{pmatrix}$$

b. Selección la variable de entrada:

$$z_1 - c_1 = c_{B_2}^T y_1 - c_1 = (1\ 5\ 0)\begin{pmatrix} -\frac{3}{4} \\ \frac{5}{8} \\ \frac{25}{8} \end{pmatrix} - 2 = 3$$

$$z_3 - c_3 = c_{B_2}^T y_3 - c_3 = (1\ 5\ 0)\begin{pmatrix} 0 \\ \frac{1}{2} \\ \frac{3}{2} \end{pmatrix} + 3 = \frac{11}{2}$$

$$z_5 - c_5 = c_{B_2}^T y_5 - c_5 = (1\ 5\ 0)\begin{pmatrix} \frac{1}{4} \\ \frac{1}{8} \\ \frac{5}{8} \end{pmatrix} + 0 = \frac{11}{2}$$

$$z_6 - c_6 = c_{B_2}^T y_6 - c_6 = (1\ 5\ 0)\begin{pmatrix} -\frac{1}{2} \\ \frac{1}{4} \\ -\frac{3}{4} \end{pmatrix} - 0 = \frac{5}{2}$$

Como se puede ver el problema ha terminado, ya que no hay ningún coste reducido negativo.

La solución óptima del problema es:

$$x^* = (0\ \ 6\ \ 0\ \ 7\ \ 0\ \ 0\ \ 29)$$

Y el valor de la función:

$$z^* = 41$$

Resolución problema dual

Para resolver el problema dual se aplica, como se vio más arriba, lo siguiente:

$$x^* = c_{B_2}^T B_2^{-1} = (1\ 5\ 0)\begin{pmatrix} \frac{1}{4} & -\frac{1}{2} & 0 \\ \frac{1}{8} & \frac{1}{4} & 0 \\ \frac{5}{8} & -\frac{3}{4} & 1 \end{pmatrix} = (\frac{7}{8}\ \frac{3}{4}\ 0)$$

Interpretación de la solución

Interpretación de la solución. De la solución se puede advertir lo siguiente:

- El problema es factible (como ya se había anticipado en la respuesta del segundo apartado) puesto que la solución no contiene valores positivos para alguna variable artificial. En particular, de hecho ni siquiera hay variables artificiales en el problema.

- La solución es única porque los costes reducidos son todos estrictamente positivos. Eso significa que cualquier cambio en la base implicaría un decremento neto en el valor de la función objetivo, mientras que el caso de soluciones múltiples ocurre cuando una cantidad infinita de soluciones tiene el mismo valor de la función objetivo.

- El valor de la función objetivo está acotado porque siempre se pudo aplicar la regla de salida con al menos, alguna variable básica.

Interpretación de los recursos. De los recursos se advierte:

- En la solución óptima sobran hasta 29 unidades del tercer recurso como lo atestigua el valor óptimo de la variable de holgura x_7. Es decir, se trata de un recurso sobrante.

- La solución del problema dual, nos indica que por cada una unidad que crece la variable x_2, la función objetivo crece $\frac{7}{8}$ unidades; que por cada una unidad que crece la variable x_4, la función objetivo crece $\frac{3}{4}$ unidades y que por cada una unidad que crece la variable x_7, la función objetivo crece 0 unidades.

2.6. Resolución gráfica

Una forma alternativa para la resolución de problemas de programación lineal, es aplicar la resolución gráfica. Uno de los problemas de este método, es que cada variable representa una dimensión en el plano. Por este motivo, este método se suele aplicar sólo cuando el problema tiene dos variables, ya que es fácil de dibujar. Con tres variables también se puede resolver un problema mediante este método, aunque es bastante complicado de representar a mano. En el caso, de que el problema tenga más de tres variables, es imposible aplicar este método puesto, que es imposible representar más de tres dimensiones. Los pasos que se deben seguir para representar un problema de programación lineal en forma gráfica, son los siguientes:

1. Dibujar un sistema de ejes cartesianos con tantas dimensiones como variables de decisión aparezcan.
2. Representar cada restricción como una región. En el caso de dos variables, cada restricción será representada como una línea, y en caso de tres variables, cada restricción será representada como un plano.
3. Calcular la intersección de todas las regiones. La intersección entre dos restricciones podrá ser un punto (cuando haya dos variables) o una línea (cuando hay tres variables). La región formada por la unión de las intersecciones entre las restricciones, se llamará *región factible*.
4. Evaluar la función objetivo en los puntos extremos de la región factible y elijo aquella que optimiza *Z* (función de optimización).

Es importante destacar, que sólo nos interesan las soluciones que quedan encuadradas dentro del primer cuadrante, ya que todas las variables de decisión deben ser mayores o iguales a 0.

Por supuesto, mediante este método también se puede detectar si un problema tiene solución única, tiene un número infinito de soluciones o si es infactible. Podremos detectar estos casos cuando:

- **Solución única**: El problema tendrá solución única, si es un único punto de la región factible, el que permite optimizar el valor de la función objetivo.
- **Número elevado de soluciones**: El problema tendrá muchas soluciones, cuando al evaluar dos puntos extremos de la región factible en la solución, ambos ofrezcan el mismo valor de la función de optimización. La solución serán todos aquellos puntos de la recta que une los puntos (en el caso de que el problema tenga dos variables) o el plano formado por las rectas (en el caso de que el problema tenga tres variables), cuya evaluación en la función objetivo es igual.
- **Infactible**: El problema será infactible cuando no sea posible encontrar una región factible entre esas restricciones.

A continuación, se va a ver un ejemplo, paso por paso, de resolución de un problema a través del método de resolución gráfica.

2.6.1. Ejemplo de resolución gráfica

El problema de programación lineal que se va a resolver es el siguiente:

$$\max Z = 2x + 10y$$
$$x + y \geq 5$$
$$-7x + 10y \leq 50$$
$$2x + y \leq 10$$
$$x, y \geq 0$$

Procedemos ahora a aplicar los pasos comentados en el apartado anterior:

1. Se representan unos ejes cartesianos, con dos dimensiones:

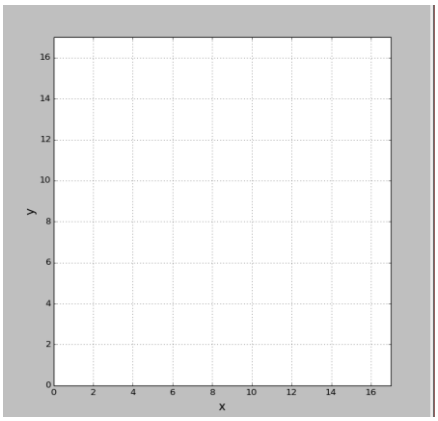

Ilustración 1: Paso 1 de representación gráfica

2. Se representa cada restricción como una región (cada región se marca en rojo, la coincidencia entre regiones, aparece en rojo más oscuro, la región final se verá mejor en el siguiente paso):

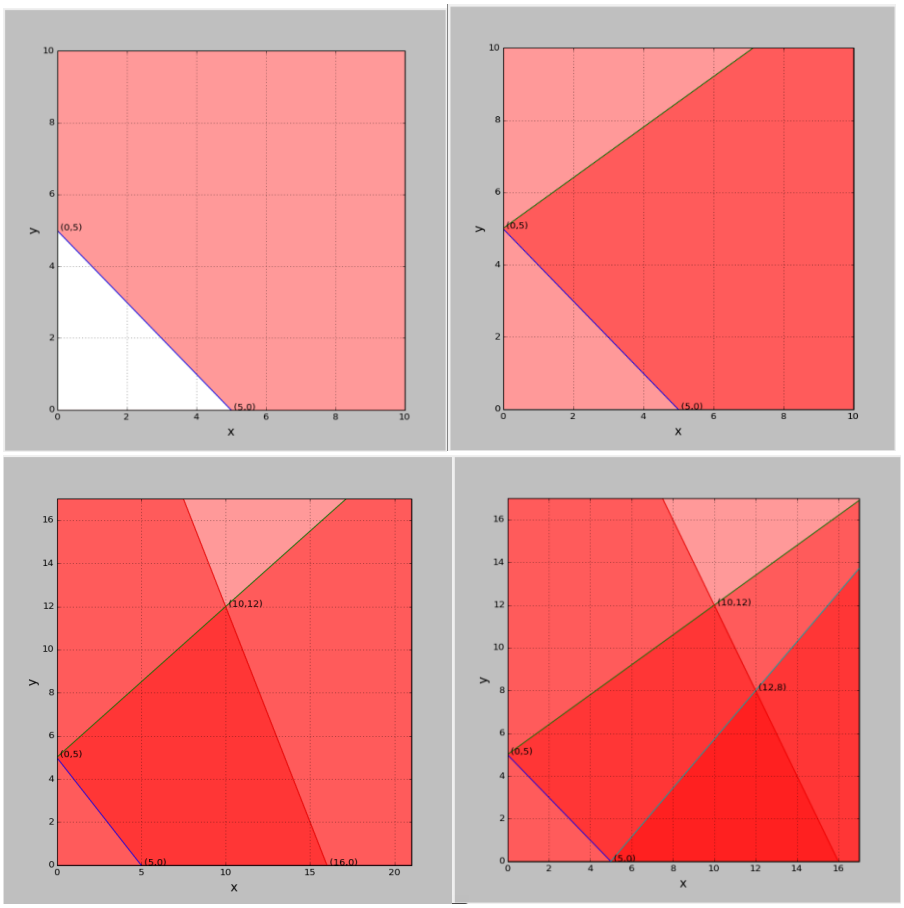

Ilustración 2: Paso 2 de representación gráfica

3. Calcular la región factible, es decir, la intersección de todas las restricciones (se ve en amarillo en la siguiente figura).

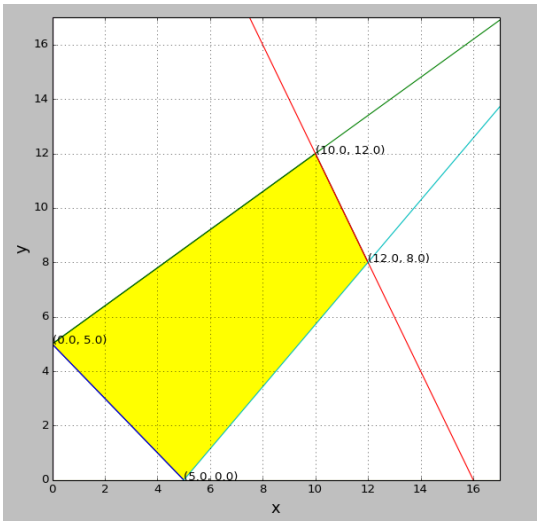

Ilustración 3: Paso 3 de representación gráfica

4. Evaluar la función objetivo en los puntos extremos de la región factible:

Puntos Extremos de la región factible:

- A (0,5)
- B (10,12)
- C (12,8)
- D (5,0)

Evaluación de la función en cada punto:

$$\max Z = 2x + 10y$$
$$Z(A) = 2 * 0 + 10 * 5 = 50$$
$$Z(B) = 2 * 10 + 10 * 12 = 140$$
$$Z(C) = 2 * 12 + 10 * 8 = 104$$
$$Z(D) = 2 * 5 + 10 * 0 = 10$$

Como se puede ver el punto que optimiza la función, es el punto B, luego la solución del problemas es x=10, y = 12.

Por supuesto, si aplicamos otro método de resolución, sobre este problema, como pudiera ser el método *Simplex*, el resultado obtenido va a ser el mismo.

2.7. Estudio de *software* similar

Si buscamos en la web, podemos encontrar varios sistemas que ofrecen de manera *online* servicios similares al sistema desarrollado de resolución de problemas de programación lineal. El objetivo de este punto, es ir analizando uno por uno, alguno de los sistemas encontrados, y finalmente, hacer una comparativa de todos ellos, y del sistema desarrollado en este trabajo final de grado.

2.7.1. Simplex Method Tool

Esta herramienta, se puede encontrar en el siguiente enlace:

http://www.zweigmedia.com/RealWorld/simplex.html

La herramienta ofrece un cuadro de diálogo, donde se puede introducir el problema a resolver. El formato del problema debe ser:

Maximize/Minimize p = x+y subject to x+y <= 2, 3x+y >= 4

A continuación ofrece varios botones dentro del cuadro de diálogo que permiten: ver un ejemplo de problema (acompañado de algunas indicaciones de formato), borrar todo lo introducido y resolver el problema. Si se opta por resolver el problema, el resultado final, aparecerá en la parte intermedia del cuadro de diálogo, donde se indicará el valor óptimo de la función objetivo, y el valor de las variables de decisión. Además en la parte inferior del cuadro, aparecerá parte del desarrollo del problema, mostrando en forma de tablas algunos de los resultados obtenidos en el mismo.

Además la herramienta ofrece la posibilidad de mostrar el resultado con números decimales, enteros o *racionales*.

2.7.2. Simplex Calculator

Esta herramienta, se puede encontrar en el siguiente enlace:

http://www.mathstools.com/section/main/simplex_online_calculator

La herramienta ofrece una ventana, en la que se puede introducir el problema a resolver. La forma de introducirlo es mediante tablas, donde se debe indicar el vector de la función objetivo, la matriz de coeficientes, los signos de las restricciones y el vector de recursos.

A la izquierda de la ventana, encontramos un conjunto de botones que permiten: añadir o quitar filas y columnas a la matriz de coeficientes, obtener el problema dual, seleccionar si se quiere minimizar o maximizar la función objetivo y si se quiere resolver el problema utilizando fracciones o mediante decimales. Por último, también se ofrece un botón para resolver el problema. En caso de pulsarlo, la ventana muestra en la parte superior una tabla con los resultados de la primera iteración, y ofrece la posibilidad de seguir viendo paso a paso el desarrollo, o pasar directamente a la solución. Al llegar a la última iteración, se muestra en la parte inferior la solución óptima del problema introducido.

Esta herramienta ofrece también una pestaña llamada "*theory*", en la que se pueden ver explicaciones teóricas de temas relacionados con la programación lineal.

También ofrece la posibilidad de descargar la herramienta, para el escritorio y como versión para *Android*.

2.7.4. Simplex me

Esta herramienta, se puede encontrar en el siguiente enlace:

http://www.simplexme.com/es/

Esta herramienta ofrece dos posibilidades de introducción del problema. La primera es mediante un cuadro de diálogo, en el que se van siguiendo los pasos indicados. Primero pide el número de variables y restricciones que tendrá el problema, y a continuación, muestra una tabla donde se deben indicar los coeficientes de la función objetivo y si hay que maximizarla o minimizarla, la matriz de coeficientes y el vector de recursos. Además ofrece la posibilidad de obtener la solución primal o la dual. Al pulsar en paso siguiente, muestra el problema introducido y la solución del mismo, y da la posibilidad de enviarla al *email* o descargarla como PDF.

La segunda posibilidad de entrada es la de introducir un problema mediante un archivo que se sube a la web. El resto del proceso es el mismo que en el caso anterior. El problema de esta opción es que no se indica cuál debe ser el formato del problema en el archivo.

2.7.5. PHP Simplex

Esta herramienta, se puede encontrar en el siguiente enlace:

http://www.phpsimplex.com/simplex/simplex.htm?l=es

Esta herramienta ofrece un primer cuadro de diálogo, en el que se debe indicar si se quiere obtener la solución mediante *Simplex*, o la resolución gráfica. Además se debe indicar el número de restricciones y variables del problema.

En el caso de indicar método *Simplex*, a continuación aparece una tabla a rellenar, con el vector de la función objetivo, el tipo de función objetivo, la matriz de coeficientes y el vector de recursos. Si se pulsa en "continuar", empieza a mostrar el desarrollo del problema, y aquí se puede elegir entre ir viendo el desarrollo paso a paso o llegar directamente a la solución final.

Si se decide optar por la solución gráfica, en ese momento se restringe el número de variables a introducir a dos. El resto del proceso de introducción del problema, es el mismo, y al pulsar en continuar, muestra la solución gráfica.

En todo momento, permite guardar el problema en un archivo y descargarlo. Además también ofrece un manual de uso.

2.7.6. Easy Calculation

Esta herramienta, se puede encontrar en el siguiente enlace:

https://www.easycalculation.com/operations-research/simplex-method-calculator.php

En esta herramienta nos encontramos con un cuadro de diálogo, en el que introducir el problema. Lo primero que hay que introducir es la función a optimizar, en la forma:

Maximize/Minimize p = x+y

A continuación, se deben introducir las restricciones, en la forma:

x+y <= 2, 3x+y >= 4

Por último, se debe indicar si se quiere obtener la solución con número decimales, *racionales* o enteros, y a cuántos decimales redondear.

Una vez completados estos campos, nos encontramos con tres botones que permiten mostrar un ejemplo de problema, borrar todo lo introducido o resolver el problema. En caso de optar por resolver el problema, primero nos encontraremos con la solución al problema, y debajo de la misma, unas tablas que muestran los valores obtenidos en cada iteración durante el desarrollo.

Esta herramienta es muy parecida a la analizada en el punto 2.6.1, aunque presenta una interfaz más amigable.

2.7.7. Linear programming calculator

Esta herramienta, se puede encontrar en el siguiente enlace:

http://www.wolframalpha.com/widget/widgetPopup.jsp?p=v&id=1e692c6f72587b2cbd3e7be018fd8960&title=Linear%20Programming%20Calculator&theme=blue

Esta herramienta, permite elegir primeramente si se quiere maximizar la función objetivo, y a continuación indicar la función de la siguiente manera:

12x+40y

Después se indican las restricciones, en un formato muy similar:

x+y<=16

Y por último, ofrece un botón "*submit*", que muestra la solución. Al mostrar la solución, se indica, el problema a resolver, la solución, y en caso de haber solo dos variables, la solución gráfica.

El problema de esta herramienta, es que restringe el número de restricciones a 5, limitando también el número de variables, porque hay que indicar además que cada variable debe ser mayor o igual a 0.

2.7.8. Excel

Aunque no es una herramienta *online*, Excel, permite resolver problemas de programación lineal. A continuación se van a comentar brevemente los pasos para resolver un problema de programación lineal, utilizando el método *Simplex* en Excel.

1. Instalar el *solver* de Excel.
2. Se introducen en una línea, en diferentes celdas, los coeficientes, de la función objetivo.
3. Además en una celda, se utiliza la función "SUMAPRODUCTO", con las celdas de la función. Esta celda será el valor de la función objetivo.
4. Se introduce la matriz de coeficientes, con un coeficiente en cada celda.
5. Se introduce el vector de recursos, con un recurso en cada celda.
6. Se abre la ventana de parámetros del *solver* y se indica, si hay que maximizar o minimizar, la celda de la función objetivo, y las restricciones, introduciendo la parte izquierda de la restricción, el signo y el recurso. En el campo "Cambiando las celdas", se introducen las celdas, donde debe aparecer la función objetivo.
7. Antes de cerrar esta ventana, accedemos a opciones, y seleccionamos, "Adoptar modelo lineal" y "Adoptar no negativos" y pulsamos "Aceptar".
8. Pulsamos ahora en "Resolver", y la hoja de cálculo, se ha actualizado con los valores de la solución y de la función objetivo.

Hay que indicar que este mismo servicio es ofrecido por *LibreOffice*, con unos pasos muy similares.

2.7.9. Comparativa

En este apartado se va a mostrar una tabla que compara las características de los distintos sistemas analizados, además de introducir el sistema *SimplexSolver*, que es el que se ha desarrollado en el presente trabajo y que en apartados posteriores se verá más en profundidad. En la siguiente tabla, se marcará con ✓, aquellas características que la herramienta presente, y con ✗, aquellas que no.

Herramientas / Características	Simplex Method Tool	Simplex Calculat.	Simplexme	PHP Simplex	Easy Calculat	Linear Program. Calculat.	Excel	SimplexSolver
Introducir problema por diálogo	✓	✓	✓	✓	✓	✓	✓	✗
Introducir problema por archivo	✗	✗	✓	✗	✗	✗	✗	✓
Solución directa	✓	✓	✓	✓	✓	✓	✓	✓
Desarrollo del problema	✓	✓	✗	✓	✓	✗	✗	✓
Solución dual	✗	✓	✗	✗	✗	✗	✗	✓
Solución gráfica	✗	✗	✗	✓	✗	✓	✗	✓
Solución Exacta (números racionales)	✓	✓	✗	✗	✓	✓	✗	✓
Guardar solución en archivo	✗	✗	✓	✓	✗	✗	✗	✓
Versión Android	✗	✓	✗	✗	✗	✗	✗	✗
Versión escritorio	✗	✓	✗	✗	✗	✗	✗	✓
Manual de uso	✗	✓	✗	✓	✗	✗	✓	✓

Tabla 1: Tabla comparativa de herramientas

Capítulo 3
Objetivos

3. Objetivos del proyecto

El propósito del proyecto será desarrollar un sistema *software*, en el lenguaje *Python*, que permita a los usuarios trabajar sobre problemas de programación lineal. El sistema debe permitir no sólo el estudio y comprensión de la solución de los problemas y de sus distintas formas, sino también visualizar la forma en la que se llega hasta ella. Más concretamente, los objetivos del sistema serán los siguientes:

- El sistema será capaz de resolver problemas de programación lineal, introducidos por el usuario, aplicando el método *Simplex*, proporcionando la solución final de forma directa, y sin mostrar cada uno de los pasos. En caso del problema, ser infactible, el sistema lo indicará de igual manera.

- El sistema será capaz de resolver problemas de programación lineal, introducidos por el usuario, de manera gráfica, siempre que el número de variables de decisión sea igual a dos, y por tanto, se permita la representación de la solución en un sistema de dos ejes cartesianos.

- El sistema permitirá la aplicación de cada uno de los pasos del método *Simplex*, proporcionando la salida o solución de cada uno de dichos pasos.

- El sistema contará con una librería que permita llamadas, a cada uno de los pasos del método *Simplex*.

- El sistema será capaz de calcular y mostrar el problema dual, de un problema introducido por el usuario.

- El sistema será capaz de resolver el problema dual, de un problema introducido por el usuario, siempre que el problema primal tenga una solución óptima. En caso del problema, ser infactible, el sistema lo indicará de igual manera.

Capítulo 4
Análisis

4. Análisis

Durante este apartado se va a discutir como se ha desarrollado el sistema. En primer lugar, se va a realizar un análisis, incluyendo la especificación de requisitos tanto funcionales como no funciones, y la descripción de los casos de uso, que se acompañará con un diagrama de casos de uso.

A continuación, se pasará a comentar el diseño del sistema. En esta parte, se realizará primero una descripción de la arquitectura global del sistema, para luego entrar más en detalle realizando un diagrama de clases, y comentando cada una de ellas. Por último se analizarán las estrategias, llevadas a cabo durante el desarrollo, enlazando los pasos a seguir con los métodos que resuelven cada una de las necesidades.

4.1. Especificación de requisitos

Aunque en el apartado 4.3, se entrará más en detalle en las partes que forman el sistema, se puede decir que el mismo, está dividido en dos partes principales. Por un lado, está la librería *Simplex.py*, que ofrece múltiples posibilidades para aplicar ciertos pasos del método *Simplex*, así como de la solución gráfica, y otros servicios para resolver cálculos matemáticos generales. Por otra parte, está el programa *SimplexSolver*. Este programa, que por supuesto utiliza la librería *Simplex.py*, permite resolver problemas de programación lineal que se le pasan al mismo (en un formato concreto) aplicando tanto el método *Simplex*, como el método de resolución gráfica.

Los requisitos se van a dividir en dos partes. Por un lado se muestran los requisitos funcionales y por otro los no funcionales. Ambos se van a mostrar en una tabla que cuenta con los siguientes campos:

- **Identificador**: este permite identificar unívocamente el requisito. Su formato es el siguiente:

  RF-XXX o **RNF-XXX**

  Donde:

  - **RF**: indica requisito funcional.
  - **RNF**: indica requisito no funcional.
  - **XXX**: será un número de tres cifras.
- **Título**: es el título que se le da al requisito.
- **Descripción**: es una breve descripción de lo cubre el requisito.

4.1.1. Requisitos funcionales

Estos requisitos son aquellos que indican, qué debe hacer el sistema. Se van a dividir en requisitos de *SimplexSolver* y requisitos de la librería *Simplex.py*, para ir desde las funcionalidades más generales a las más específicas que presentará el sistema en su conjunto.

4.1.1.1. Requisitos SimplexSolver

Al utilizar este programa, la librería *Simplex.py*, los requisitos de *SimplexSolver* no son más que una ampliación o selección de los servicios que ofrece la librería.

Id requisito	RF-001
Título	Resolución de problema
Descripción	El sistema debe poder resolver un problema de programación lineal (tanto en forma normal estándar, como en cualquier forma), que se le pase como argumento en un archivo de texto plano.

Tabla 2: RF-001

Id requisito	RF-002
Título	Solución en consola
Descripción	El sistema debe poder mostrar la solución por consola de un problema de programación lineal, y en una ventana interactiva la solución gráfica.

Tabla 3: RF-002

Id requisito	RF-003
Título	Solución en archivo
Descripción	El sistema debe poder escribir la solución de un problema de programación lineal, en un archivo que se indica como argumento. También guardar en una imagen la solución gráfica.

Tabla 4: RF-003

Id requisito	RF-004
Título	Solución dual
Descripción	El sistema debe poder mostrar la solución del problema dual, a uno introducido como argumento.

Tabla 5: RF-004

Id requisito	RF-005
Título	Formato del problema
Descripción	El problema que se pasa como argumento, debe tener el siguiente formato (por ejemplo): min -1 -2 1 1 = 6 3 1 = 12 1 1 <= 16

Tabla 6: RF-005

Id requisito	RF-006
Título	Solución directa
Descripción	El sistema debe poder mostrar la solución del problema de manera directa.

Tabla 7: RF-006

Id requisito	RF-007
Título	Mostrar desarrollo
Descripción	El sistema debe poder mostrar todo el desarrollo del problema hasta llegar a la solución.

Tabla 8: RF-007

Id requisito	RF-008
Título	Solución gráfica
Descripción	El sistema debe poder resolver gráficamente un problema de programación lineal, siempre y cuando tenga sólo dos variables.

Tabla 9: RF-008

4.1.1.2. Requisitos Simplex.py

A continuación, se muestran los requisitos de la librería *Simplex.py*.

Id requisito	RF-009
Título	Procesar archivo
Descripción	La librería debe permitir procesar un archivo que contenga un problema de programación lineal, diferenciando cada una de las partes del mismo.

Tabla 10: RF-009

Id requisito	RF-010
Título	Forma estándar
Descripción	La librería debe permitir transformar un problema de programación lineal a su forma normal estándar.

Tabla 11: RF-010

Id requisito	RF-011
Título	Variables iteración
Descripción	La librería debe permitir calcular cuáles son las variables que forman cada iteración del método *Simplex*.

Tabla 12: RF-011

Id requisito	RF-012
Título	Variables que no están en la iteración
Descripción	La librería debe permitir calcular cuáles son las variables que no forman parte de cada iteración del método *Simplex*.

Tabla 13: RF-012

Id requisito	RF-013
Título	Base iteración
Descripción	La librería debe permitir calcular y mostrar la base de cada iteración del método *Simplex*.

Tabla 14: RF-013

Id requisito	RF-014
Título	Solución de iteración
Descripción	La librería debe permitir calcular y mostrar el valor de las variables para una iteración (solución de la iteración).

Tabla 15: RF-014

Id requisito	RF-015
Título	Valor del vector de recursos
Descripción	La librería debe permitir calcular y mostrar cuáles son las variables del vector de recursos que pertenecen a la iteración (variables de la iteración)

Tabla 16: RF-015

Id requisito	RF-016
Título	Valor de la función
Descripción	La librería debe permitir calcular y mostrar el valor que toma la función a optimizar para cada iteración.

Tabla 17: RF-016

Id requisito	RF-017
Título	Valores de *y*
Descripción	La librería debe permitir calcular y mostrar los valores de *y* para cada iteración.

Tabla 18: RF-017

Id requisito	RF-018
Título	Regla de entrada
Descripción	La librería debe permitir calcular y mostrar los valores de la regla de entrada para cada iteración.

Tabla 19: RF-018

Id requisito	RF-019
Título	Continuidad del problema
Descripción	La librería debe poder calcular y mostrar si el problema ha terminado o continúa, al finalizar cada iteración.

Tabla 20: RF-019

Id requisito	RF-020
Título	Variable de entrada en siguiente iteración
Descripción	La librería debe permitir calcular y mostrar cuál es la variable que entra en la iteración siguiente.

Tabla 21: RF-020

Id requisito	RF-021
Título	Regla de salida
Descripción	La librería debe permitir calcular y mostrar los valores de la regla de salida de cada iteración.

Tabla 22: RF-021

Id requisito	RF-022
Título	Variable que sale en la iteración
Descripción	La librería debe permitir calcular y mostrar cuál es la variable que sale en cada iteración.

Tabla 23: RF-022

Id requisito	RF-023
Título	Resolver iteración
Descripción	La librería debe permitir resolver una iteración completa del método *Simplex*, mostrando todos los pasos.

Tabla 24: RF-023

Id requisito	RF-024
Título	Columnas matriz identidad
Descripción	La librería debe permitir saber si una matriz, tiene columnas de la matriz identidad.

Tabla 25: RF-024

Id requisito	RF-025
Título	Añadir variables artificiales a la matriz de coeficientes
Descripción	La librería debe permitir añadir variables artificiales al problema, a través de la introducción de columnas de la matriz identidad, a la matriz de coeficientes.

Tabla 26: RF-025

Id requisito	RF-026
Título	Coeficiente variables artificiales
Descripción	La librería debe permitir calcular, cuál es el coeficiente para una variable artificial, en una función dada. Este coeficiente, será equivalente a $-\infty$, que es el coeficiente con el que se añaden las variables a la función objetivo.

Tabla 27: RF-026

Id requisito	RF-027
Título	Añadir variables artificiales a la función
Descripción	La librería debe permitir añadir variables artificiales a una función de maximización o minimización.

Tabla 28: RF-027

Id requisito	RF-028
Título	Omitir comentarios
Descripción	La librería debe permitir omitir los comentarios del tipo "//" y "#" de un archivo dado.

Tabla 29: RF-028

Id requisito	RF-029
Título	Transformación de la función
Descripción	La librería debe poder transformar una función de minimización a una de maximización.

Tabla 30: RF-029

Id requisito	RF-030
Título	Cambio de signo
Descripción	La librería debe poder transformar un signo de”>=” o “<=” a su opuesto. También se permite el cambio de los signos “<” y ”>”, aunque no sean válidos para los problemas de programación lineal.

Tabla 31: RF-030

Id requisito	RF-031
Título	Transformación de restricciones
Descripción	La librería debe poder transformar las restricciones, en caso de que el recurso sea negativo, para que deje de serlo.

Tabla 32: RF-031

Id requisito	RF-032
Título	Solución a un problema
Descripción	La librería debe permitir resolver un problema de programación lineal mostrando todos los pasos.

Tabla 33: RF-032

Id requisito	RF-033
Título	Cálculo de solución dual
Descripción	La librería debe poder calcular la solución dual de un problema de programación lineal.

Tabla 34: RF-033

Id requisito	RF-034
Título	Determinar tipo de solución
Descripción	La librería debe poder determinar si el problema tiene solución única, infinitas soluciones, o es no acotado.

Tabla 35: RF-034

Id requisito	RF-035
Título	Representar restricción
Descripción	La librería debe permitir representar una restricción de manera gráfica, y siempre que tenga dos variables.

Tabla 36: RF-035

Id requisito	RF-036
Título	Puntos de intersección
Descripción	La librería debe permitir calcular los puntos de intersección entre restricciones.

Tabla 37: RF-036

Id requisito	RF-037
Título	Región factible
Descripción	La librería debe permitir calcular los puntos extremos de la región factible.

Tabla 38: RF-037

Id requisito	RF-038
Título	Puntos que optimizan la función
Descripción	La librería debe permitir calcular cuál de los puntos de la región factible optimiza la función.

Tabla 39: RF-038

Id requisito	RF-039
Título	Valor de la función en un punto
Descripción	La librería debe permitir calcular el valor de la función, para un punto dado.

Tabla 40: RF-039

Id requisito	RF-040
Título	Puntos enteros
Descripción	La librería debe permitir calcular los puntos enteros que están dentro de la región factible.

Tabla 41: RF-040

Id requisito	RF-041
Título	Transformar restricción en función
Descripción	La librería debe permitir transformar una restricción en una función para ser representada.

Tabla 42: RF-041

Id requisito	RF-042
Título	Eliminar puntos
Descripción	La librería debe permitir eliminar puntos negativos y repetidos, dentro de una lista de puntos.

Tabla 43: RF-042

Id requisito	RF-043
Título	Punto cumple restricción
Descripción	La librería debe permitir comprobar si un punto cumple una restricción.

Tabla 44: RF-043

Id requisito	RF-044
Título	Punto pertenece a región factible
Descripción	La librería debe permitir comprobar si un punto pertenece a la región factible.

Tabla 45: RF-044

Id requisito	RF-045
Título	Ordenar puntos
Descripción	La librería debe permitir ordenar un conjunto de puntos en sentido horario.

Tabla 46: RF-045

Id requisito	RF-046
Título	Mostrar solución gráfica
Descripción	La librería debe permitir mostrar la solución gráfica de un problema de programación lineal, siempre que este tenga solo dos variables.

Tabla 47: RF-046

Id requisito	RF-047
Título	Mostrar solución en ventana
Descripción	La librería debe permitir mostrar la solución gráfica de un problema de programación lineal en una ventana interactiva.

Tabla 48: RF-047

Id requisito	RF-048
Título	Mostrar solución en ventana
Descripción	La librería debe permitir guardar la solución gráfica de un problema de programación lineal como una imagen.

Tabla 49: RF-048

Id requisito	RF-049
Título	Uso de números *racionales*
Descripción	Todo el sistema deberá utilizar, en todos los cálculos que realice, números *racionales* en lugar de números reales.

Tabla 50: RF-049

4.1.2. Requisitos no funcionales

Estos requisitos son aquellos, que debe presentar el equipo para poder hacer uso del *software* desarrollado. No encontramos muchos requisitos de este tipo, ya que no hay ningún tipo de restricciones de hardware para la ejecución.

Id requisito	RNF-001
Título	Sistema operativo
Descripción	El *software* debe poder utilizarse en cualquier sistema operativo.

Tabla 51: RNF-001

Id requisito	RNF-002
Título	Python
Descripción	El equipo debe tener instalado *Python* en su versión 3.4.

Tabla 52: RNF-002

Id requisito	RNF-003
Título	Numpy
Descripción	Se debe instalar en *Python* el paquete *numpy*.

Tabla 53: RNF-003

Id requisito	RNF-004
Título	Scipy
Descripción	Se debe instalar en *Python* el paquete *scipy*.

Tabla 54: RNF-004

Id requisito	RNF-005
Título	Matplotlib
Descripción	Se debe instalar en *Python* el paquete *matplotlib*.

Tabla 55: RNF-005

Id requisito	RNF-006
Título	ipython
Descripción	Se debe instalar en *Python* el paquete *ipython*.

Tabla 56: RNF-006

4.2. Casos de uso

En este apartado se van a visualizar y describir los casos de uso de todo el sistema en su conjunto (incluyendo librería *Simplex.py* y *SimplexSolver*). Estos casos de uso permiten definir claramente el sistema, viendo cuáles son las funcionalidades que éste presenta.

4.2.1. Agentes

Entendiendo por agente, todo aquel usuario que pueda hacer uso del sistema, se va a distinguir entre dos tipos de agentes o roles:

- **Alumno**: Se entiende que este agente será una persona interesada en resolver un problema de programación lineal.
- **Desarrollador**: Se entiende que este agente será una persona que necesite de alguno de los servicios que la librería *Simplex.py* ofrece, para el desarrollo de un *software* determinado.

Por supuesto, ambos roles no son excluyentes, es decir, un alumno en un momento dado podría necesitar de algún servicio de la librería o por el contrario, un desarrollador, podría estar interesado en resolver un problema.

4.2.2. Diagrama de casos de uso

Mediante la herramienta *StarUML*, que permite el desarrollo de diagramas UML (*Unified Modeling Language*) de todo tipo, para el modelado de sistemas, se han

desarrollado los diagramas de casos de uso. Se han desarrollado dos diagramas, uno por cada uno de los agentes comentados en el apartado anterior, y cubriendo así todas las posibles funcionalidades del sistema. Las siguientes figuras, muestran dichos diagrama:

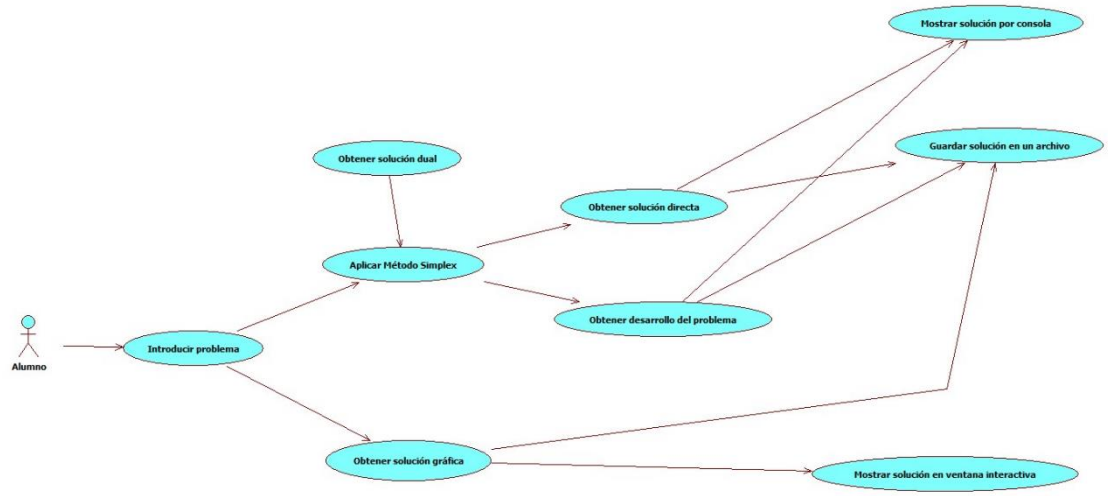

Ilustración 4: Diagrama de casos de uso (I). Contiene casos de uso 1, 2, 3, 4, 5, 6, 7,8 y 9.

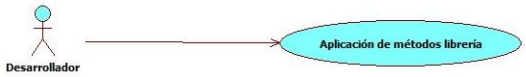

Ilustración 5: Diagrama de casos de uso (II). Contiene caso de uso 10.

4.2.3. Descripción de los casos de uso

En los siguientes apartados, se van a describir en detalle cada uno de los casos de uso presentes en los diagramas de las ilustraciones 4 y 5. En cada uno de ellos, además de una pequeña descripción, se incluirá una tabla que contendrá la siguiente información:

- **Identificador:** identifica unívocamente un caso de uso. Presentará el siguiente formato:

  CU-001

  Donde:

  - **CU:** son unas siglas que significan “Caso de uso”.
  - **XXX:** es un número que identifica el caso de uso.
- **Nombre:** se corresponde con el nombre del caso de uso que aparece en el diagrama de casos de uso.
- **Actores:** identifica cuáles son los actores que cubren ese caso de uso.
- **Objetivo:** identifica cuál será el objetivo de dicho caso de uso.
- **Precondiciones:** determina cuáles son las condiciones necesarias para que el caso de uso pueda llevarse a cabo.
- **Secuencia:** indica los pasos a seguir para llevar a cabo el caso de uso.

4.2.3.1. Introducir problema

Para la resolución de un problema de programación lineal, el alumno primero debe introducir el problema mediante un archivo de texto plano, y en el formato correcto. Una vez escrito el problema en un archivo, se podrá proceder a la resolución del mismo, mediante el uso de *SimplexSolver*.

Identificador	**CU-001**
Nombre	Introducir problema
Actores	Alumno
Objetivos	Introducir problema de programación lineal en el formato apropiado para que pueda ser resuelto.
Precondiciones	• Crear un archivo vacío de texto plano.
Secuencia	1- Crear un archivo vacío de texto plano. 2- Escribir problema en el siguiente formato: min -1 -2 1 1 = 6 3 1 = 12 1 1 <= 16 3- Guardar archivo con el problema en la misma localización que se encuentre *SimplexSolver*.

Tabla 57: CU-001

4.2.3.2. Aplicar Método Simplex

Una vez se ha escrito el problema correctamente en un archivo, el alumno puede utilizar *SimplexSolver*, para que el método *Simplex* se aplique de forma automática sobre el problema introducido. También se podría optar por aplicar manualmente cada uno de los pasos del método *Simplex*, utilizando los métodos de la librería *Simplex.py*.

Identificador	**CU-002**
Nombre	Aplicar método *Simplex*
Actores	Alumno
Objetivos	Aplicar el método *Simplex* sobre un problema de programación lineal.
Precondiciones	Usando *SimplexSolver*: • Tener un archivo de texto plano con el problema en el formato apropiado. Usando la librería *Simplex.py*: • Tener la librería instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	<u>Secuencia 1:</u> 1- Pasar el archivo como parámetro de *SimplexSolver*. <u>Secuencia 2:</u> 2- Importar librería *Simplex.py*. 3- Aplicar uno por uno los pasos del método *Simplex*, utilizando los métodos de la librería.

Tabla 58: CU-002

4.2.3.3. Obtener solución dual

El alumno además de obtener la solución del problema, podría necesitar conocer también la solución del problema dual, al que se ha introducido. Si se indica a *SimplexSolver* que se quiere obtener la solución del problema dual, esta se mostrará también, a continuación de la solución del problema primal. También podría obtenerse esta solución mediante la correcta aplicación de algunos métodos de la librería *Simplex.py.*

Identificador	**CU-003**
Nombre	Obtener solución dual
Actores	Alumno
Objetivos	Obtener la solución del problema dual, al introducido.
Precondiciones	Usando *SimplexSolver*: • Tener un archivo de texto plano con el problema en el formato apropiado. Usando la librería *Simplex.py*: • Tener la librería instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	Secuencia 1: 1- Pasar el archivo como parámetro de *SimplexSolver*. 2- Indicar como parámetro, que se quiere obtener la solución del problema dual. Secuencia 2: 1- Importar librería *Simplex.py*. 2- Aplicar uno por uno los pasos necesarios para obtener la solución dual, utilizando los métodos de la librería.

Tabla 59: CU-003

4.2.3.4. Obtener solución gráfica

Una vez está el problema escrito en un archivo de texto plano y en el formato correcto, se podría optar por obtener la solución gráfica del mismo. Hay que resaltar que la solución gráfica solo puede obtenerse cuando el problema tenga como máximo dos variables. Para obtener esta solución, simplemente hay que indicárselo mediante un parámetro a *SimplexSolver*. En este caso, se mostrará tanto la resolución gráfica como la solución aplicando el método *Simplex*. También podría hacerse aplicando ciertos métodos de la librería *Simplex.py*, aunque como en el caso anterior resulta mucho más complejo.

Identificador	CU-004
Nombre	Obtener solución gráfica
Actores	Alumno
Objetivos	Alcanzar la solución gráfica de un problema de programación lineal
Precondiciones	Usando *SimplexSolver*: • Tener un archivo de texto plano con el problema en el formato apropiado. Usando la librería *Simplex.py*: • Tener la librería instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	Secuencia 1: 1- Pasar el archivo como parámetro de *SimplexSolver*. 2- Ejecutar *SimplexSolver* indicando por parámetro, que se quiere obtener la solución gráfica. Secuencia 2: 1- Importar librería *Simplex.py*. 2- Aplicar uno por uno los pasos necesarios para obtener la solución gráfica, usando los métodos de la librería.

Tabla 60: CU-004

4.2.3.5. Obtener solución directa

El alumno, una vez ejecutado *SimplexSolver* podrá visualizar la solución del problema, de manera directa. Es decir, se le mostrará simplemente el valor de las variables y de la función objetivo. En caso de optar, por querer obtener la solución del problema dual (ver en el apartado 4.2.3.3), también se mostrará la misma.

Identificador	CU-005
Nombre	Obtener solución directa
Actores	Alumno
Objetivos	Obtener solo la solución del problema
Precondiciones	Tener un archivo de texto plano con el problema en el formato apropiado.
Secuencia	1- Pasar como parámetro el archivo con el problema a *SimplexSolver*. 2- Ejecutar *SimplexSolver*, sin indicar por parámetro que se quiere obtener el desarrollo completo del problema.

Tabla 61: CU-005

4.2.3.6. Obtener desarrollo del problema

Al ejecutar *SimplexSolver* sobre un problema, podría indicarse que se quiere visualizar el desarrollo completo del problema. Ello significa, que se verán uno por uno todos los pasos del método *Simplex* aplicados, sobre el problema introducido por parámetro. Finalmente se mostrará la solución del mismo, como ocurre en el CU-005.

Al igual que en el caso de uso CU-002, se podría llegar a dicha solución también, aplicando los métodos de la librería *Simplex.py*, de la forma y orden apropiado.

Identificador	**CU-006**
Nombre	Obtener desarrollo del problema
Actores	Alumno
Objetivos	Visualizar de manera completa, la aplicación del método *Simplex*.
Precondiciones	Usando *SimplexSolver*: • Tener un archivo de texto plano con el problema en el formato apropiado. Usando la librería *Simplex.py*: • Tener la librería instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	Secuencia 1: 1- Pasar como parámetro el archivo con el problema a *SimplexSolver*. 2- Indicar por parámetro que se quiere visualizar el desarrollo completo del problema. 3- Ejecutar *SimplexSolver*. Secuencia 2: 1- Importar librería *Simplex.py*. 2- Aplicar uno por uno los pasos del método *Simplex*, utilizando los métodos de la librería.

Tabla 62: CU-006

4.2.3.7. Mostrar solución en ventana interactiva

Cuando el alumno, opta por obtener la solución gráfica (cuando sea posible obtenerla), tendrá la opción de que la solución gráfica aparezca en una ventana interactiva, que permitirá al alumno, moverse sobre la figura representada. También se pude acercar, alejar, guardar… Al igual que en el caso de uso CU-004, se podría visualizar dicha ventana, aplicando los métodos de la librería *Simplex.py*, en la forma y orden apropiado.

Identificador	CU-007
Nombre	Mostrar solución en ventana interactiva
Actores	Alumno
Objetivos	La solución gráfica se muestre en una ventana que permita, interacción del alumno con la figura representada.
Precondiciones	Usando *SimplexSolver*: • Tener un archivo de texto plano con el problema en el formato apropiado. Usando la librería *Simplex.py*: • Tener la librería instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	Secuencia 1: 1- Pasar el archivo como parámetro de *SimplexSolver*. 2- Ejecutar *SimplexSolver* indicando mediante parámetro, que se quiere obtener la solución gráfica, y sin indicar archivo de destino de la solución. Secuencia 2: 1- Importar librería *Simplex.py*. 2- Aplicar uno por uno los pasos necesarios para obtener la solución gráfica, usando los métodos de la librería.

Tabla 63: CU-007

4.2.3.8. Mostrar solución por consola

Cuando el alumno ejecute el *SimplexSolver*, para obtener la solución de un problema que se le pasa en un archivo, podrá visualizar por consola el resultado, tanto si es simplemente la solución, como si es el desarrollo completo. En el caso de usar la librería, para aplicar los pasos del método *Simplex*, las ejecuciones de los métodos siempre se mostrarán por consola.

Identificador	CU-008
Nombre	Mostrar solución por consola
Actores	Alumno
Objetivos	Bien sea la solución únicamente, o el desarrollo completo del problema, sea visualizado por consola, al ejecutar.
Precondiciones	Tener un archivo de texto plano con el problema en el formato apropiado.
Secuencia	1- Pasar como parámetro el archivo con el problema a *SimplexSolver*. 2- Indicar por parámetro que se quiere visualizar el desarrollo completo del problema. 3- Ejecutar *SimplexSolver*, sin indicar archivo de destino de la solución.

Tabla 64: CU-008

4.2.3.9. Escribir solución en archivo

Cuando el alumno ejecute *SimplexSolver*, puede optar por guardar la solución del problema, en un archivo, para así poder volver a ella cuando desee. Al ejecutar *SimplexSolver*, e indicar que se quiere guardar la información en un archivo, aparecerá en la localización donde esté *SimplexSolver*, un archivo nuevo, con el nombre que se indicó, que contendrá la solución. Si además se indica que se quiere la solución gráfica, se creará una imagen con la solución. Si se obtiene la solución de manera manual, esta opción no está disponible, es decir aplicando los métodos de la librería, a no ser que el usuario, vuelque directamente la solución en un archivo.

Identificador	**CU-009**
Nombre	Escribir solución en archivo
Actores	Alumno
Objetivos	Obtener un archivo con la solución.
Precondiciones	Tener un archivo de texto plano con el problema en el formato apropiado.
Secuencia	1- Pasar como parámetro el archivo con el problema a *SimplexSolver*. 2- Indicar por parámetro si se quiere visualizar el desarrollo completo del problema. 3- Indicar por parámetro si se quiere obtener la solución gráfica. 4- Indicar por parámetro, que se quiere guardar la solución en un archivo, y el nombre del archivo. 5- Ejecutar *SimplexSolver*.

Tabla 65: CU-009

4.2.3.10. Aplicación de métodos librería

Un desarrollador, podría necesitar de alguno de los métodos presentes en la librería *Simplex.py*, para añadir o complementar alguna funcionalidad del *software* que esté desarrollando.

Identificador	**CU-010**
Nombre	Aplicación de métodos librería
Actores	Desarrollador
Objetivos	Utilizar la librería *Simplex.py* para otros desarrollos
Precondiciones	La librería *Simplex.py* debe estar instalada en el sistema. En caso de no usar *setuptools* o *pip* para la instalación, la librería debe estar en la misma localización que el *software* donde se estén incluyendo los métodos de la misma.
Secuencia	1- Importar librería *Simplex.py*. 2- Aplicar métodos de la librería que se consideren oportunos.

Tabla 66: CU-009

4.3. Diseño

En este apartado se va a definir, cuál es la estructura del sistema. En primer lugar se va a definir la arquitectura del sistema, viendo de manera general las partes del sistema. Seguidamente, se incluirá un diagrama de clases, para ver como se ha

implementado la arquitectura descrita, y además se hará una pequeña descripción de cada clase.

A continuación se describirán las estrategias seguidas para desarrollar las diferentes funcionalidades, así como la correspondencia entre métodos y pasos de las estrategias seguidas.

Al final se presentan los manuales para facilitar el uso del sistema.

4.3.1. Arquitectura del sistema

No se ha seguido ningún patrón específico de diseño, puesto que no hay ninguno que encaje con el tipo de desarrollo que aquí se va a describir. Por ejemplo, no tendría ningún sentido, aplicar un MVC (*Model View Controller*), ya que no contamos con ninguno de los elementos propios de este patrón. Por lo tanto, se ha optado por descomponer en funcionalidades, para ir construyendo el sistema.

En el siguiente diagrama, se puede ver la diferenciación de cada una de las partes de las que se compone el sistema.

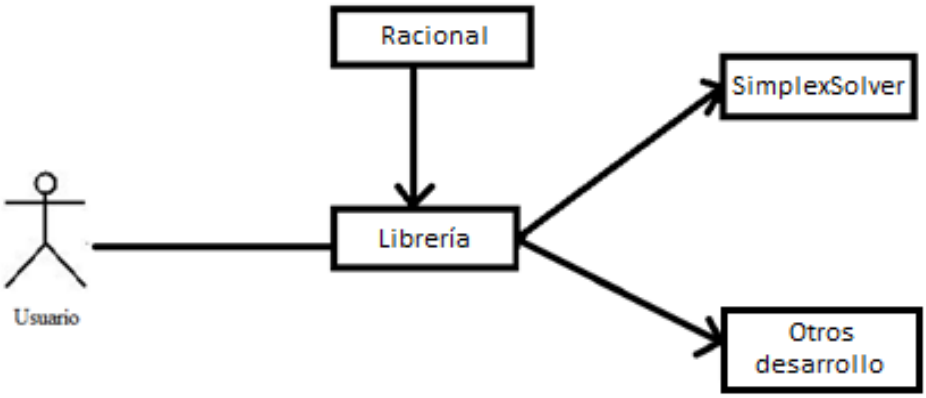

Ilustración 6: Arquitectura del sistema

- ***Racional*:** esta parte del sistema, será una implementación de los números racionales, puesto que como se comenta en el apartado 2.4 y en el requisito RF-049, se optará porque todo el sistema funcione con números racionales, ya que así la solución del sistema será exacta.
- ***Librería***: esta parte del sistema constituirá el núcleo del mismo. Contendrá toda la lógica para resolver problemas de programación lineal, así como para realizar algunos cálculos matemáticos de carácter general.
- **SimplexSolver**: esta parte del sistema será la encargada de llamar a la librería para que se resuelva un problema que ha recibido, así como de mostrar la solución o guardarla, en función de lo que indique el usuario.
- **Otros desarrollos**: esto no se refiere a un módulo del sistema en sí, sino a una posibilidad que ofrece el mismo. Es posible que un usuario, utilice la librería, para completar servicios de su propio *software*.

4.3.2. Diagrama de clases

El diagrama de clases permite representar las clases que componen el sistema. En él se podrá ver los atributos de cada clase, así como los métodos que componen cada una. Así mismo se pueden ver las relaciones entre las clases que conforman el sistema.

La representación queda de la siguiente forma:

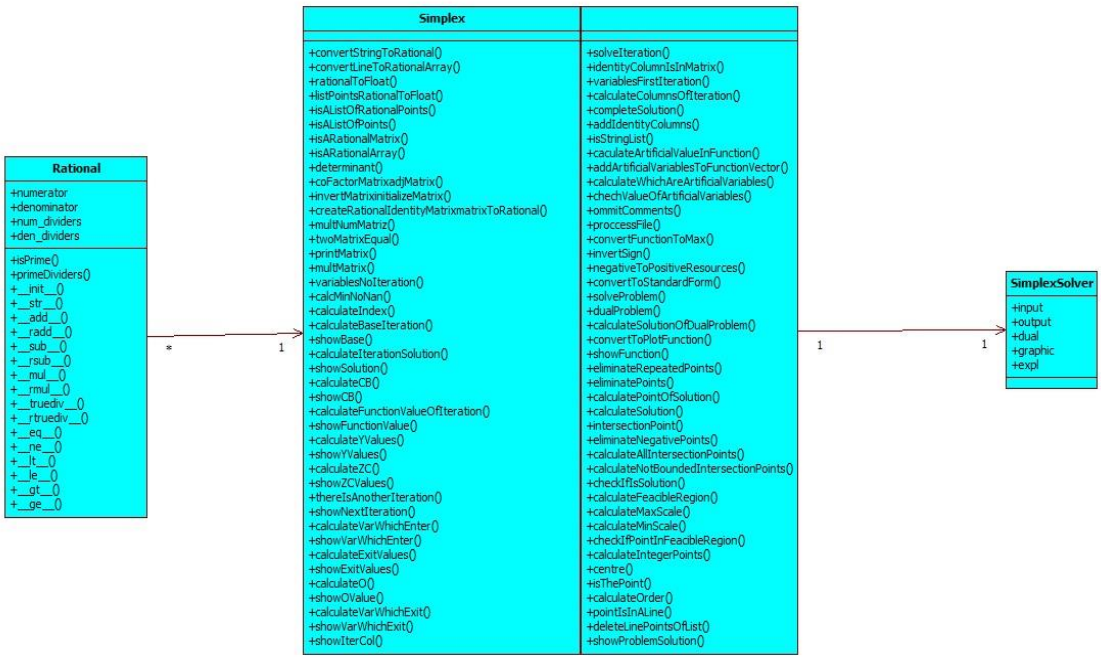

Ilustración 7: Diagrama de clases

Como se puede ver, el sistema queda bastante simple, puesto que la complejidad del mismo no viene en su arquitectura, sino en el desarrollo del algoritmo para la resolución de problemas. Además se puede comprobar que el desarrollo final de las clases, encaja perfectamente con la arquitectura diseñada en el apartado anterior.

4.3.3. Descripción de clases

A continuación se van a describir, cada una de las clases que se pueden ver en el diagrama de clases.

4.3.3.1. Clase Rational.py

Esta clase, será la que implementa los números *racionales*. Cada objeto de la clase, tendrá un numerador y un denominador. El denominador será 1, cuando se trate de números enteros. Además también se calcularán los divisores primos del numerador y del denominador, para así poder simplificar el número *racional*, y agilizar las operaciones realizadas con estos objetos. El introducir esta clase, ha ralentizado los tiempos de ejecución, pero se ha asumido este "inconveniente", para mejorar en la exactitud del problema. Esta clase contará con métodos que permitirán sobrescribir todas las operaciones que se realizan con números reales para así poder usar los *racionales* en su lugar.

4.3.3.2. Clase Simplex.py

Esta clase constituye el núcleo del sistema puesto que es la librería que contiene todos los métodos, que se van a utilizar, para aplicar el método *Simplex* sobre un problema de programación lineal, además de ofrecer otros servicios, para resolver cálculos matemáticos en general. Esta librería utilizará la clase *rational.py* para todos sus cálculos, y será llamada por *SimplexSolver.py* para resolver problemas de manera automática. Además también contiene métodos que permiten obtener la solución gráfica, siempre y cuando el problema tenga dos variables. A pesar de pertenecer a la misma clase, la librería, podría dividirse en cuatro partes bien diferenciadas:

1. **Operaciones con rational**: el conjunto de métodos que compone esta parte de la librería, son utilizados para realizar ciertas operaciones con los objetos *rational*. Muchos de los métodos se utilizan para hacer comprobaciones en otros métodos, de que los parámetros introducidos son correctos.
2. **Operaciones con matrices**: esta parte de la librería contiene algunas de las típicas operaciones con matrices que existen, como invertir matrices, multiplicar matrices… En la librería *numpy*, de *Python*, existen métodos que realizan estas operaciones, pero han tenido que ser redefinidas para que funcionen con objetos *rational*, conllevando esto una perdida evidente en la velocidad de ejecución.
3. **Simplex**: Los métodos de esta parte de la librería, son los que se utilizan para desarrollar la solución de un problema de programación lineal, mediante el uso del método *Simplex*. Se encuentran aquí métodos para calcular la regla de entrada, la regla de salida, la solución de una iteración, etc.
4. **Solución gráfica**: en esta parte de la librería aparecen los métodos que permiten calcular la solución gráfica de un problema. Por tanto, se encuentran aquí métodos que permiten representar restricciones, buscar la región factible de un conjunto de intersecciones, calcular cuáles son los puntos óptimos de esa región, etc.

4.3.3.3. Clase *SimplexSolver*

Esta clase es la encargada de resolver un problema de programación lineal que recibe por parámetro. No contiene métodos propios, sino que importa la librería *Simplex.py*, para utilizar algunos de sus métodos.

Dentro de este subsistema, se analizan los parámetros que se reciben, para saber las opciones que hay que tomar, en la solución a mostrar. Los parámetros que puede recibir, le indicarán:

- El archivo de entrada que contiene (este parámetro es obligatorio).
- Si se quiere mostrar o no el desarrollo del problema.
- Si se quiere volcar o no el resultado en un archivo.
- Si se quiere obtener la solución del problema dual.
- Si se quiere obtener la solución gráfica del problema.

Una vez analizados los parámetros, se actuará en consecuencia. Luego dentro de esta clase, lo que se hace simplemente es analizar los parámetros recibidos y llamar a los métodos de la librería en función de lo que los parámetros indiquen.

4.3.4. Estrategias

En este apartado se va a describir, cuáles han sido las estrategias seguidas para el desarrollo de cada funcionalidad.

4.3.4.1. Estrategia números racionales

A continuación, se va a describir cuál ha sido la estrategia definida para crear una clase que implemente los números *racionales*. Como en apartados anteriores, ya se ha comentado que simplemente un objeto *rational* se forma por un numerador y un denominador, además de dos listas de enteros, que serán los divisores primos de numerador y denominador, se va a comentar cuál ha sido la estrategia seguida para calcular cada una de las funcionalidades de las que disponen estos objetos. Podemos dividir estas funcionalidades en dos grandes grupos. La primera sería la simplificación de un número *racional*, y la segunda los operadores que dan lugar a las operaciones que se pueden realizar con estos números.

1. Calcular divisores primos del numerador y el denominador.
2. Ver que números primos son divisores del numerador y del denominador.
3. Dividir el numerador y el denominador, por los divisores primos comunes, tantas veces como se pueda.

4.3.4.1.1. Correspondencia entre pasos y métodos de *rational*

A continuación se muestra la correspondencia entre los métodos de la clase *rational*, que se utilizan para simplificar y los pasos seguidos en el apartado anterior.

Paso de simplificación	Método de *rational*
1. Calcular divisores primos del numerador y el denominador.	• primeDividers ○ isPrime
2. Ver que números primos son divisores del numerador y del denominador.	• simplify
3. **Dividir el numerador y el denominador, por los divisores primos comunes, tantas veces como se pueda.**	

Tabla 67: Correspondencia entre pasos y métodos de rational

4.3.4.1.2. Operadores

En este apartado se va a explicar cuál ha sido la estrategia seguida para implementar los operadores de los objetos *rational*. Se va a realizar la explicación mediante un ejemplo:

Operadores	Estrategia de la implementación
Suma	$\frac{a}{b}+\frac{c}{d}=Simplificar(\frac{ad+bc}{bd})$
Suma con un entero	

Resta	$a + \frac{c}{d} = Simplificar(\frac{ad+c}{d})$ $\frac{a}{b} - \frac{c}{d} = Simplificar(\frac{ad-bc}{bd})$
Resta con un entero	$a - \frac{c}{d} = Simplificar(\frac{ad-c}{d})$
Multiplicación	$\frac{a}{b}.\frac{c}{d} = Simplificar(\frac{ac}{bd})$
Multiplicación con un entero	$a.\frac{c}{d} = Simplificar(\frac{ac}{d})$
División	$\frac{a}{b}:\frac{c}{d} = Simplificar(\frac{ad}{bc})$
División con un entero	$a:\frac{c}{d} = Simplificar(\frac{ad}{c})$
Igual	$\frac{a}{b} = \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad = bc$
Distinto	$\frac{a}{b} \neq \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad \neq bc$
Mayor que	$\frac{a}{b} > \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad > bc$
Menor que	$\frac{a}{b} < \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad < bc$
Mayor o igual	$\frac{a}{b} \geq \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad \geq bc$
Menor o igual	$\frac{a}{b} \leq \frac{c}{d}$ -> $Simplificar(\frac{a}{b}), Simplificar(\frac{c}{d})$ ->$ad \leq bc$

Tabla 68: Operadores

No se va a indicar la correspondencia entre operadores y métodos de la clase *rational*, porque ya quedó indicado en el apartado 4.3.2.1, que cada operación tenía un método implementado que sobrescribía las operaciones que se pueden realizar con los números reales.

4.3.4.2. Estrategia Simplex

Se va a describir aquí cuál ha sido la estrategia seguida, para obtener un sistema que pueda resolver un problema de programación lineal. En este caso, la estrategia, es bastante sencilla, puesto que lo único que se ha hecho ha sido ir siguiendo en el orden correcto los pasos del método *Simplex*. Los pasos seguidos son los siguientes:

1. Transformación del archivo recibido, para obtener el problema en sí.
2. Transformar el problema a forma normal estándar.

3. Añadir variables artificiales, en caso de ser necesario.
4. Obtener la base de la primera iteración (será la matriz identidad).
5. Calcular la solución de la iteración.
6. Calcular el valor de la función para dicha iteración.
7. Calcular los valores de la regla de entrada.
8. Comprobar si el problema continúa, o ha terminado.
9. Obtener variable que entra en la siguiente iteración.
10. Calcular los valores de la regla de salida.
11. Ver qué variable sale, o si el problema es no acotado.
12. Mejora de la solución.
13. Obtener solución final.
14. Obtener problema dual.
15. Obtener solución dual.

4.3.4.2.1. Correspondencia entre pasos del método *Simplex* y *Simplex*.py

En la siguiente tabla, se va a mostrar la relación existente entre cada uno de los pasos del apartado anterior y los métodos de *Simplex.py*, que realizan dicho paso.

Paso del método Simplex	Método de Simplex.py
1. Transformación del archivo recibido, para obtener el problema en sí.	• proccessFile ○ ommitComments
2. Transformar el problema a forma normal estándar.	• convertToStandardForm ○ convertFunctionToMax ○ invertSign ○ negativeToPositiveResources
3. Añadir variables artificiales, en caso de ser necesario.	• identityColumnIsInMatrix • addIdentityColumns • addArtificialVariablesToFunctionVector • calculateArtificialValueInFunction
4. Obtener la base de la primera iteración (será la matriz identidad).	• variablesFirstIteration • identityColumnIsInMatrix
5. Calcular la solución de la iteración.	• calculateIterationSolution • showSolution
6. Calcular el valor de la función para dicha iteración.	• calculateCB • showCB calculateFunctionValueOfIteration showFunctionValue
7. Calcular los valores de la regla de entrada.	• calculateYValues • showYValues • calculateZC • showZCValues
8. Comprobar si el problema continúa, o ha terminado.	• thereIsAnotherIteration • showNextIteration
9. Obtener variable que entra en la siguiente iteración.	• calculateVarWhichEnter • showVarWhichEnter

10. Calcular los valores de la regla de salida.	• calculateExitValues • showExitValues • calculateO • showOValue
11. Ver que variable sale, o si el problema es no acotado.	• calculateVarWhichExit • calculateIndex • showVarWhichExit
12. Mejora de la solución.	• calculateColumnsOfIteration • calculateBaseIteration • showBase • Vuelta al paso 5
13. Obtener solución final.	• checkValueOfArtificialVariables • completeSolution
14. Obtener problema dual.	• dualProblem
15. Obtener solución dual.	• calculateSolutionOfDualProblem
Aplicar todos los pasos de una iteración.	• solveIteration
Aplicar todos los pasos para un problema.	• solveProblem

Tabla 69: Correspondencia entre pasos método simplex y Simplex.py

4.3.3.3. Estrategia solución gráfica

En este apartado, se va a describir, cuáles han sido los pasos seguidos, para obtener la solución gráfica del problema. Algunos de estos pasos, obviamente, coinciden con los aplicados para calcular la solución mediante el método *Simplex*. Los pasos seguidos, son los que siguen:

1. Transformación del archivo recibido, para obtener el problema en sí.
2. Se transforma cada restricción en una función, es decir, se despeja la variable independiente.
3. Se representan las funciones correspondientes a cada restricción.
4. Se calcula el punto de corte, entre cada una de las restricciones y los ejes, y entre cada una de las restricciones entre sí.
5. Se calcula qué puntos de los anteriores, cumplen todas las restricciones. Estos puntos serán los puntos extremos de la región factible.
6. Se ordenan los puntos de la región factible en el sentido de las agujas del reloj.
7. Se representa la región factible y los puntos extremos.
8. Se calcula qué punto/s extremo/s de la región factible optimizan la función.
9. Se calculan los puntos enteros dentro de la región factible.
10. Se muestra la solución final.

4.3.4.3.1. Correspondencia entre pasos de solución gráfica y Simplex.py

En la siguiente tabla, se va a mostrar la relación existente entre cada uno de los pasos para aplicar la resolución gráfica del apartado anterior y los métodos de *Simplex.py*, que realizan dicho paso.

Paso resolución gráfica	Método de Simplex.py
1. Transformación del archivo recibido, para obtener el problema en sí.	• proccessFile • ommitComments
2. Se transforma cada restricción en una función, es decir, se despeja la y (variable independiente).	• convertToPlotFunction
3. Se representan las funciones correspondientes a cada restricción.	• showFunction
4. Se calcula el punto de corte, entre cada una de las restricciones y los ejes, y entre cada una de las restricciones entre sí. Solo se mantienen los positivos.	• calculateAllIntersectionPoints ○ intersectionPoint ○ eliminateNegativePoints ○ eliminateRepeatedPoints ○ eliminatePoints
5. Se calcula qué puntos de los anteriores, cumplen todas las restricciones. Estos puntos serán los puntos extremos de la región factible.	• calculateFeacibleRegion ○ checkIfIsSolution
6. Se ordenan los puntos de la región factible en el sentido de las agujas del reloj.	• calculateOrder ○ centre ○ isThePoint
7. Se representa la región factible y los puntos extremos.	• Este paso se realiza la representar el problema completo
8. Se calcula qué punto/s extremo/s de la región factible optimizan la función.	• calculateSolution • calculatePointOfSolution
9. Se calculan los puntos enteros dentro de la región factible.	• calculateIntegerPoint ○ pointIsInALine ○ deleteLinePointsOfList
10. Se muestra la solución final.	• calculateIfPointInFeacibleRegion
Representar un problema de forma completa	• showProblemSolution ○ calculateMaxScale ○ calculateMinScale

Tabla 70: Correspondencia entre pasos de solución gráfica y Simplex.*py*

4.3.5. Manuales

Durante el desarrollo, también se han implementado dos manuales, mediante la herramienta *ipython notebook*. Esta herramienta, es una herramienta web, que además de permitir la inclusión de texto e imágenes, ofrece celdas, que permiten la ejecución de código *Python*. De esta forma, el usuario, no solo podrá leer las explicaciones de cada manual, sino visualizar la ejecución de ejemplos, los cuáles puede modificar a su antojo, para ver las variaciones en el resultado. Los dos manuales implementados, son un

manual de usuario y un manual del desarrollador. Ambos manuales, están pensados para complementar y explicar las funcionalidades del sistema.

Para acceder a estos manuales, simplemente accedemos por línea de comandos a la carpeta donde se encuentre tanto el sistema, como los manuales y ejecutamos el siguiente comando:

localizaciónPython*/Scripts/ipython notebook

Por localizaciónPython, se entiende que será la localización de *Python*, por ejemplo, C:/python34/. Una vez ejecutado este comando, se abrirá el navegador, y aparecerá una lista de todo el contenido de la carpeta en la que nos encontramos. En este momento, se hace doble clic sobre el manual que se desee, y aparecerá el mismo.

4.3.5.1. Manual del usuario

En este manual, se explica cuáles son las distintas formas de ejecutar el subsistema *SimplexSolver*, así como todas las funcionalidades que ofrece. Además de una descripción detallada de cada una de ellas, se pueden ver ejemplos de ejecución y el resultado que se obtiene.

Se implementan todas las funcionalidades del mismo:

- Visualización de la solución.
- Resolución problema dual.
- Visualización del desarrollo completo del problema.
- Visualización de la solución gráfica.
- Carga de la solución en un archivo.

4.3.5.2. Manual del desarrollador

En este manual, se explican uno a uno, todos los métodos de la librería *Simplex.py*. En todos ellos se seguirá la misma dinámica, empezando por una explicación del método, indicando qué hace, qué recibe y qué devuelve. A continuación, se muestran ejemplos de ejecuciones del método, entre los que se incluyen tanto ejecuciones correctas, como aquellos casos en los que se introduzcan parámetros que no sean los adecuados.

Capítulo 5
Pruebas

5. Pruebas

Durante este capítulo se van a mostrar algunos de los resultados obtenidos, al ejecutar el programa. En primer lugar, se van a realizar unos casos de prueba, que cubren prácticamente la totalidad de escenarios posibles que se pueden dar en una ejecución del sistema, o al menos los más comunes.

Por último, se realizará un estudio para analizar la eficiencia del sistema de manera detallada, y se sacarán conclusiones al respecto.

5.1. Casos de prueba

En este apartado se van a mostrar algunas ejecuciones de *SimplexSolver*. Se han buscado algunos casos especiales para mostrar cómo reacciona el programa ante ellos, además también se podrá ver cuál es el formato de salida de las soluciones. Para cada caso de uso, se mostrará el problema en notación matemática, después se indicará el formato del problema en el archivo que se le debe pasar a *SimplexSolver* y, por último, se mostrará la salida del programa.

5.1.1. Caso de prueba 1

En este caso de prueba se va a introducir el problema siguiente:

$$\begin{gathered}\min Z = -2x_1 - x_2 + 3x_3 - 5x_4 \\ x_1 + 2x_2 + 2x_3 + 4x_4 \leq 40 \\ -2x_1 + x_2 - x_3 - 2x_4 \geq -8 \\ 4x_1 - 2x_2 + x_3 - x_4 \leq 10 \\ x \geq 0\end{gathered}$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
min -2 -1 3 -5
1 2 2 4 <= 40
-2 1 -1 -2 >= -8
4 -2 1 -1 <= 10
```

Ilustración 8: Archivo con problema del caso de prueba 1

Este problema, es el problema que se mostraba como ejemplo en la sección 2.5. Luego, el problema presenta solución única y ese debería ser el resultado que se muestra al ejecutar *SimplexSolver*, sobre el problema. También se calculará la solución dual:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo5.txt --du
al
SOLUTION: x* =[0 , 6 , 0 , 7 , 0 , 0 , 29] ,  z* = -41 .There is a unique soluti
on.
DUAL SOLUTION: x* =[0 , 7/8 , 0 , 3/4 , 0 , 0 , 0]
```

Ilustración 9: Ejecución caso de prueba 1

Como se puede ver la salida de la ejecución, coincide con la obtenida en el apartado 2.5. Así mismo también coincide la solución del problema dual.

Puesto que es el primer caso de prueba se va a presentar, el formato de la salida identificando cada parte. Primero se pedía la solución del problema primal, esta se muestra con la palabra "SOLUTION", a la que acompaña el valor de las variables de decisión (x^*), y el valor óptimo de la función (z^*). Respecto al conjunto de variables se muestra el valor final de las variables de decisión, así como de las variables artificiales que se hayan tenido que añadir. De esta forma el usuario puede darse cuenta de los recursos que sobren o falten en función de cada variable. Por último, también se incluye una frase que indica el tipo de solución. En este caso la solución es única, pero en otros casos de uso se verá cómo la solución puede ser de otro tipo, y así queda indicado.

Por otro lado, se requería también la solución del problema dual, esta se muestra a continuación de las palabras "DUAL SOLUTION", y nuevamente se indica el valor de todas las variables de decisión (x^*). Cuando no sea posible calcular la misma, aparecerá una indicación de ello, como se verá en los siguientes casos de prueba.

Si se quisiera ver el desarrollo completo del problema, se podría añadir el atributo - - expl, y se mostrará en consola el desarrollo del problema además de la solución.

5.1.2. Caso de prueba 2

En este caso de prueba se va a introducir el problema siguiente:

$$\begin{gathered}\max Z = 2x_1 + x_2 \\ 2x_1 + x_2 \leq 18 \\ -x_1 + x_2 \leq 8 \\ 5x_1 + 2x_2 \geq 0 \\ x \geq 0\end{gathered}$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
max 2 1
2 1 <= 18
-1 1 <= 8
5 2 >= 0
```

Ilustración 10: Archivo con problema del caso de prueba 2

Este problema, tiene como particularidad que presenta infinitas soluciones, luego esa debe ser la salida que muestre *SimplexSolver*. Se intenta calcular también el problema dual.

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo4.txt --gr
aphic --dual
SOLUTION: x* =[9 , 0 , 0 , 17 , 45 , 0] ,  z* = 18 .There are a large amount of
solutions.
DUAL SOLUTION: x* =[1 , 0 , 0 , 0 , 0 , 0]
```

Ilustración 11: Ejecución caso de prueba 2 (I)

Como se puede ver la salida de la ejecución indica que el problema tiene infinitas soluciones, y además se muestra la solución de la última iteración del problema.

También se puede ver que la solución del problema dual, sí está disponible para este tipo de problemas.

Además se va a buscar también la solución gráfica de dicho problema. *SimplexSolver* muestra lo siguiente:

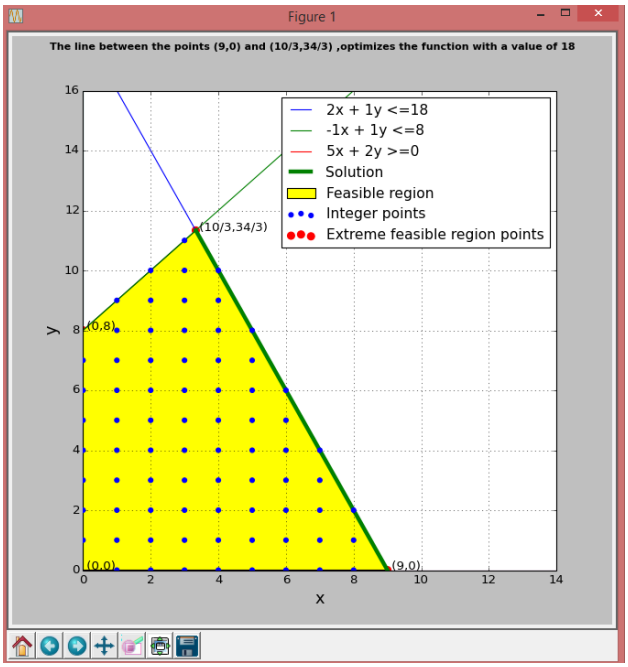

Ilustración 12: Ejecución caso de prueba 2 (II)

Como se puede ver en la solución gráfica, también se indica que el problema tiene infinitas soluciones, ya que además de mostrar que la solución es una línea de puntos, aparece una explicación en la parte superior.

5.1.3. Caso de prueba 3

En este caso de prueba se va a introducir el problema siguiente:

$$\max Z = 3x_1 + x_2$$
$$x_1 \geq 1$$
$$x_2 \leq \frac{5}{2}$$
$$x_1 + x_2 \geq \frac{3}{2}$$
$$-x_1 + x_2 \leq 1$$
$$x_2 \geq 0$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
max 3 1
1 0 >= 1
0 1 <= 5/2
1 1 >=3/2
-1 1 <= 1
0 1 >= 0
```

Ilustración 13: Archivo con problema del caso de prueba 3

En este caso, el problema introducido es no acotado, y así debería indicarlo *SimplexSolver*. Además se intenta buscar la solución dual:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo15.txt --g
raphic --dual
SOLUTION: x* =[3/2 , 0 , 1/2 , 5/2 , 0 , 5/2 , 0 , 0 , 0 , 0] ,  z* = 9/2 .The p
roblem is not bounded, because it can not exit any variable.
Dual problem solution is not available for this kind of primal problem.
```

Ilustración 14: Ejecución caso de prueba 3 (I)

La salida muestra que el problema es no acotado, y además, indica la solución de la última iteración. También se dice que el problema dual, no está disponible para este tipo de problemas.

Se busca también la solución gráfica de dicho problema:

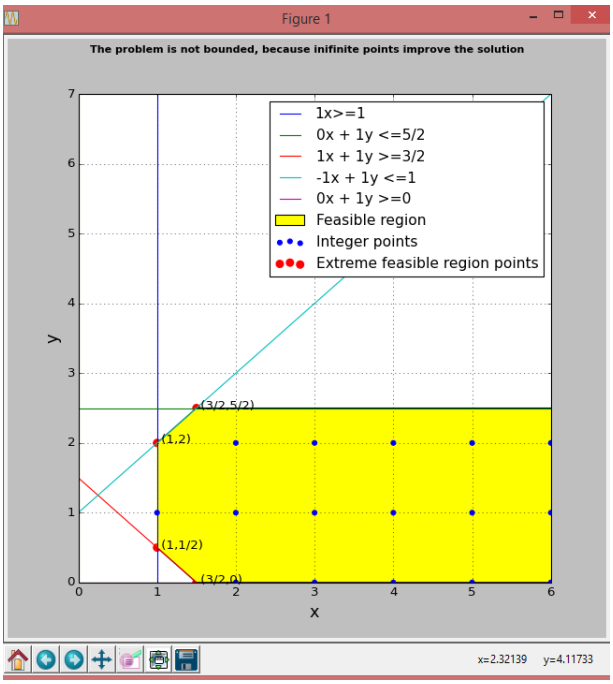

Ilustración 15: Ejecución caso de prueba 3 (II)

La solución gráfica, coincide con la salida anterior e indica que el problema es no acotado, además de con la propia solución, con una explicación.

5.1.4. Caso de prueba 4

En este caso de prueba, se va a introducir el problema siguiente:

$$\min Z = 4x_1 + x_2$$
$$2x_1 + x_2 \geq 18$$
$$-x_1 + x_2 \geq 8$$
$$5x_1 + 2x_2 \leq -10$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
min 4 1
2 1 >= 18
-1 1 >= 8
5 2 <= -10
```

Ilustración 16: Archivo con problema del caso de prueba 4

Este problema no tiene solución, y eso debe mostrar *SimlpexSolver*:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo19.txt --g
raphic --dual
SOLUTION: x* =[0 , 0 , 0 , 0 , 0 , 18 , 8 , 10] ,  z* = 740 .The problem does no
t have solution because, as we can see, the artificial variable/s [8] has/have p
ostive value.
Dual problem solution is not available for this kind of primal problem.
```

Ilustración 17: Ejecución caso de prueba 4 (I)

Como se puede ver en la salida de la ejecución, se muestra que el problema no tiene solución, y se puede ver la solución de la última iteración del problema. Además se indica que la solución dual, como es lógico no está disponible.

Al obtener la solución gráfica se obtiene lo siguiente:

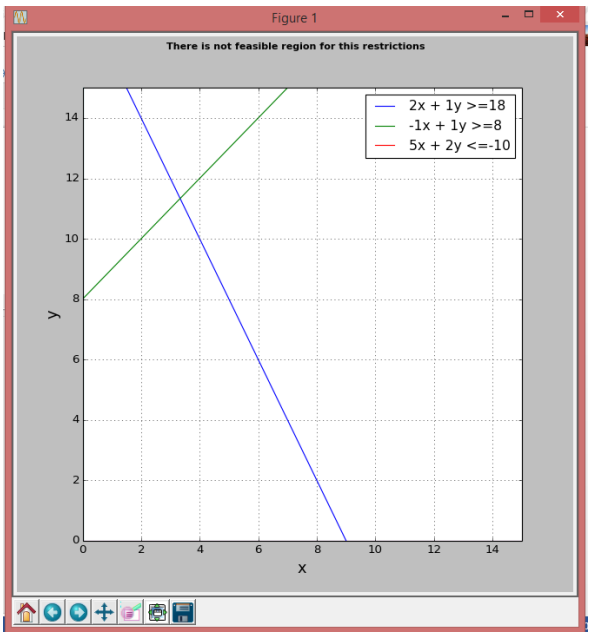

Ilustración 18: Ejecución caso de prueba 4 (II)

Así mismo también, la representación gráfica indica que no hay solución posible para este problema.

5.1.5. Caso de prueba 5

En este caso de prueba, se va a introducir el problema siguiente:

$$\max Z = x_1 + 2x_2 - 3x_3$$
$$2x_1 + x_2 - x_3 \leq 4$$
$$x_1 + x_2 + 2x_3 \geq 3$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
#function
max 1 2 -3
//first restriction
2 1 -1 <= 4
1 1 2 >= 3 # Second restriction
//End of the problem
```

Ilustración 19: Archivo con problema del caso de prueba 5

Este problema no tiene ninguna particularidad, simplemente comprueba el fichero de entrada. Este fichero contiene comentarios en diferentes lugares y de diferentes tipos. El propósito es comprobar que el fichero se procesa estén donde estén los comentarios. Se ejecuta *SimplexSolver* y la salida es la siguiente:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo1.txt
SOLUTION: x* =[0 , 4 , 0 , 0 , 1 , 0] ,  z* = 8 .There is a unique solution.
```

Ilustración 20: Ejecución caso de prueba 5

Como se puede ver, el problema se procesa sin ningún tipo de inconveniente, estén donde estén los comentarios.

5.1.6. Caso de prueba 6

En este caso de prueba se va a introducir el problema siguiente:

$$\max Z = \frac{2}{10}x_1 + \frac{8}{100}x_2$$
$$\frac{1}{10}x_1 \geq \frac{4}{10}$$
$$\frac{1}{10}x_2 \geq \frac{6}{10}$$
$$\frac{1}{10}x_1 + \frac{2}{10}x_2 \geq 2$$
$$\frac{2}{10}x_1 + \frac{1}{10}x_2 \geq \frac{17}{10}$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
max 2/10 8/100
1/10 0 >= 4/10
0 1/10 >= 6/10
1/10 2/10 >= 2
2/10 1/10 >= 17/10
```

Ilustración 21: Archivo con problema del caso de prueba 6

La particularidad de este problema es que se introducen fracciones tanto en la función objetivo como en las restricciones. El objetivo es comprobar que se procesa el archivo y se resuelve sin problemas al introducir fracciones. La salida obtenida es:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo21.txt --g
raphic
SOLUTION: x* =[20 , 0 , 8/5 , 0 , 0 , 23/10 , 0 , 3/5 , 0 , 0] ,  z* = -8 .The p
roblem is not bounded, because it can not exit any variable.
```

Ilustración 22: Ejecución caso de prueba 6 (I)

Como se puede ver el problema es resuelto de forma satisfactoria. Se busca también la solución gráfica, obteniendo el mismo resultado:

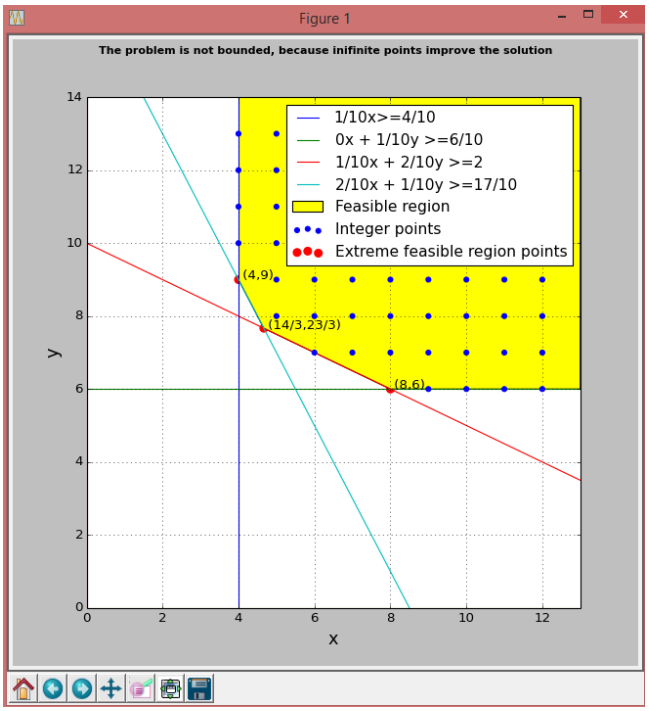

Ilustración 23: Ejecución caso de prueba 6 (II)

5.1.7. Caso de prueba 7

En este caso de prueba se va a introducir el problema siguiente:

$$\min Z = -2x_1 - 3x_2 - 4\,x_3$$
$$3x_1 + \frac{5}{2}x_2 + x_3 \leq 10$$
$$2x_1 + \frac{11}{2}x_2 + \frac{7}{2}x_3 \leq 15$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
min -2 -3 -4
3 2.5 1 <= 10
2 5.5 3.5 <= 15
```

Ilustración 24: Archivo con problema del caso de prueba 7

Este problema, como se puede ver tiene decimales. Los decimales no son permitidos, sino que hay que introducir los datos del problema mediante fracciones. *SimplexSolver* así lo indica:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo3.txt --du
al
Please introduce the problem in the correct format. Only fractions are allowed.
```

Ilustración 25: Ejecución caso de prueba 7

5.1.8. Caso de prueba 8

En este caso de prueba se va a introducir el problema siguiente:

$$\max Z = x_1 + 2x_3$$
$$2x_1 - x_2 + x_3 \leq 4$$
$$-x_1 + x_2 + 2x_3 \leq 3$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
max 1 0 2
2 -1 1 <= 4
-1 1 2 <= 3
```

Ilustración 26: Archivo con problema del caso de prueba 8

Lo que se pretende comprobar en este caso, es qué ocurre cuando se pretende buscar la solución gráfica de un problema con más de dos variables. La salida obtenida es la siguiente:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo6.txt --du
al --graphic
SOLUTION: x* =[7 , 10 , 0 , 0 , 0] ,  z* = 7 .There is a unique solution.
DUAL SOLUTION: x* =[1 , 1 , 0 , 0 , 0]
Graphic solution is only available for problems with two variables.
```

Ilustración 27: Ejecución caso de prueba 8

Como se puede ver, *SimplexSolver* indica que la solución gráfica no está disponible.

5.1.9. Caso de prueba 9

En este caso de prueba se va a introducir el problema siguiente:

$$\max\ Z = 2x_1 + 10x_2$$
$$x_1 + x_2 \geq 5$$
$$-7x_1 + 10x_2 \leq 50$$
$$2x_1 + 1x_2 \leq 32$$
$$8x_1 - 7x_2 \leq 40$$
$$x \geq 0$$

El formato de entrada del problema para *SimplexSolver*, sería el siguiente:

```
max 2 10
1 1 >= 5
-7 10 <= 50
2 1 <= 32
8 -7 <= 40
```

Ilustración 28: Archivo con problema del caso de prueba 9

Lo que se va a probar en este caso, es qué ocurre cuando se indica un fichero de salida, a la ejecución de *SimplexSolver*. La ejecución es la siguiente:

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo20.txt --o
utput salida.txt --dual --graphic
```

Ilustración 29: Ejecución caso de prueba 9

Si ahora se comprueba el directorio donde está guardado el programa, aparecerán dos archivos, uno con la solución del método *Simplex* y otro con la solución gráfica.

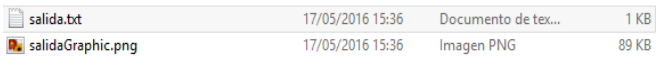

Ilustración 30: Imagen del directorio donde aparecen los archivos

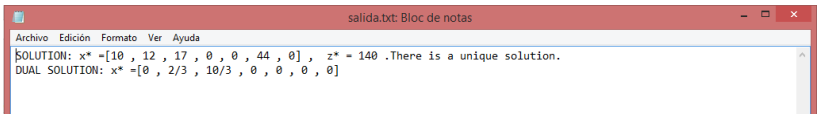

Ilustración 31: Archivo de salida con la solución del caso de prueba 9 (I)

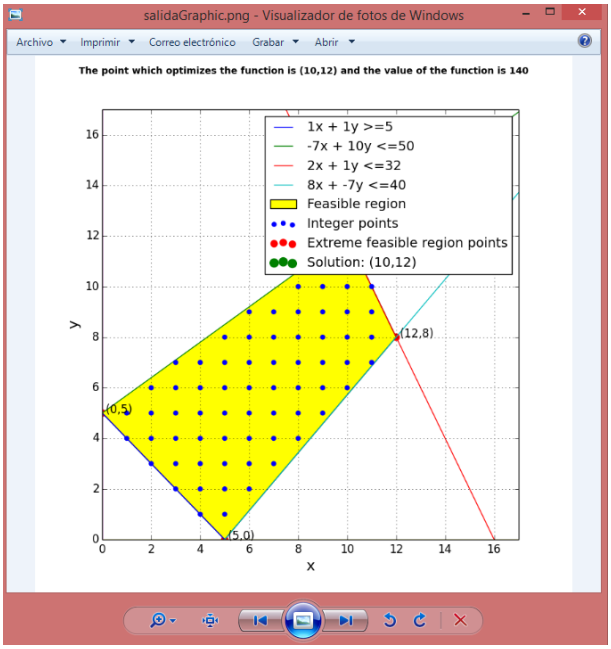

Ilustración 32: Archivo de salida con la solución del caso de prueba 9 (II)

En caso de que no se quiera la solución gráfica, este archivo no aparecerá. Por otro lado si se añade el atributo - - expl, en lugar de guardar solo la solución en el archivo, aparecerá el desarrollo del problema, en dicho archivo.

5.2. Estudio de la eficiencia

Durante varias partes del presente documento se hace alusión al rendimiento de la aplicación. Como se comenta, no se ha prestado demasiada atención al rendimiento, puesto que se ha preferido sacrificar eficiencia, en búsqueda de una mejora de la exactitud en la resolución de problemas. En definitiva, se ha sacrificado la eficiencia, para ofrecer un mejor servicio, o al menos más útil para los fines académicos del proyecto.

A pesar de todo ello, se ha querido realizar un estudio, de cómo evoluciona el comportamiento del programa en función de diferentes parámetros. Es decir, se va a comprobar cuáles son los factores que influyen en que el *software* actúe más o menos rápido.

Para el estudio se ha generado un programa (también en *Python*), que genera problemas aleatorios. Este programa recibe como parámetros el número de variables que debe tener el problema y el número de restricciones del mismo, y genera 100 archivos en el formato apropiado para ser resueltos por *SimplexSolver*. No se tiene en cuenta en este caso si los problemas serán irresolubles o tendrán solución, simplemente se generan con el fin de ejecutar *SimplexSolver* sobre ellos y comprobar tiempos de ejecución.

Por otro lado se ha generado otro código, que coja los 100 archivos generados y ejecute sobre ellos *SimplexSolver*. Se calcula en este proceso además el tiempo de ejecución. En todas las ejecuciones de *SimplexSolver*, se buscará también la solución del problema dual (si es que la hay).

Se pasa a continuación a entrar más en detalle en el estudio.

5.2.1. Aumento de variables

Durante esta parte del estudio, se fijará el número de restricciones y se irá incrementando el número de variables. Se ha decidido fijar el número de restricciones a 3, ya que así habrá cálculos con matrices 3X3, algo que dará un resultado y unos tiempos más aproximados a la realidad que si trabajamos con matrices 2x2, que obviamente será mucho más rápido.

El proceso por lo tanto, consistirá en utilizar el *software* generador de problemas, para crear problemas que tengan tres restricciones. Se empezará entonces generando, problemas con dos variables, luego con tres, y así sucesivamente hasta llegar a 7 variables. Se ha parado en este número de variables, porque el tiempo de ejecución ya comienza a ser algo alto, y además, porque es un número de variables que hace que el problema sea difícilmente resoluble a mano, luego la utilidad de *SimplexSolver* quedaría muy reducida.

En la siguiente tabla se muestran los tiempos obtenidos, al realizar esta parte del estudio, aplicando *SimplexSolver* sobre los problemas generados:

	2 Variables	3 Variables	4 Variables	5 Variables	6 Variables	7 Variables
Tiempo de ejecución de 100 problemas						
Tiempo de ejecución(ms)	7899	125703	600733	4689432	5087822	13535920
Tiempo de ejecución(s)	7,899	125,703	600,733	4689,432	5087,822	13535,920
Tiempo de ejecución por problema(media)						
Tiempo de ejecución(ms)	78,99	1257,03	6007,33	46894,32	50878,22	135359,2
Tiempo de ejecución(s)	0,07899	1,25703	6,00733	46,89432	50,87822	135,3592

Tabla 71: Resultados del estudio en función de las variables

En la siguiente gráfica se muestran los datos de la tabla anterior, para comprobar de una forma más visual, la progresión de la ejecución:

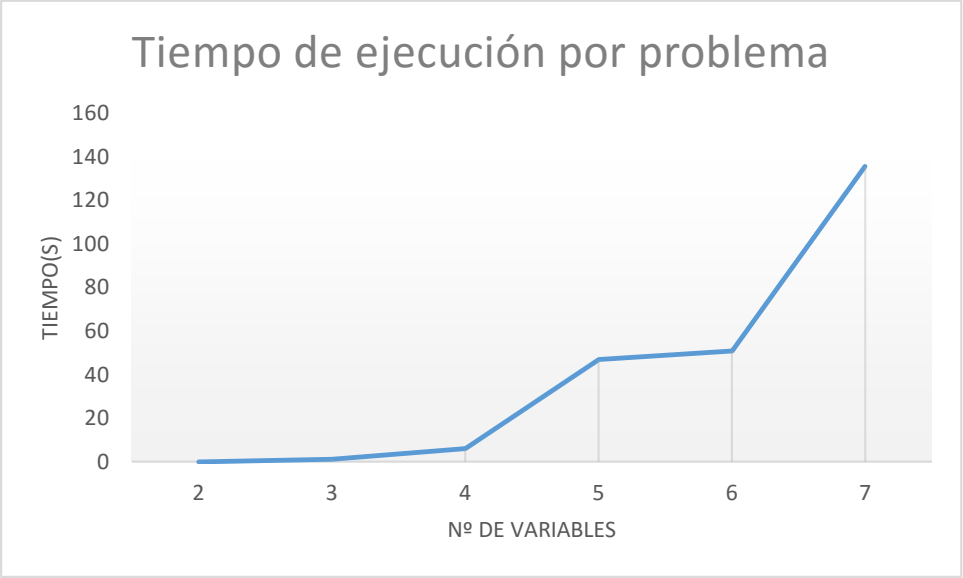

Ilustración 33: Gráfico de tiempo de ejecución en función de variables

Como se puede ver tanto en los datos como en el gráfico el tiempo de ejecución del problema comienza a dispararse cuando incluimos más de 4 variables. Al aumentar el número de variables aumenta el número de iteraciones del problema, luego es lógico que cuando el número de variables de decisión llega a números elevados como pueda ser 6 o 7, el tiempo de ejecución suba. Por supuesto, a este número de variables de decisión habría que añadir las variables de holgura y artificiales necesarias en cada caso, luego el número de iteraciones aumenta. Cuando se habla de que el número de iteraciones aumenta, realmente se quiere decir que aumenta el número de posibilidades de que haya más iteraciones, puesto que el número de variables no garantiza más iteraciones en cualquiera de los casos. Podría darse que hubiera muchas variables y al realizar la regla de entrada en la primera iteración, salgan todos los valores positivos, luego el problema habría terminado. Sería un caso de muchas variables y una iteración.

Este hecho podría explicar, como hay casos en los que al aumentar el número de variables no se producen saltos significativos de tiempo (como en aumento de 5 a 6 variables), y sin embargo en otros se produzcan saltos mayores (como en el aumento de 4 a 5 variables).

5.2.2. Aumento de restricciones

Esta parte del estudio será la opuesta a la parte anterior. En este caso se va a fijar el número de variables y lo que va a ir aumentando será el número de restricciones.

Se ha decidido en este caso fijar el número de variables de decisión a tres. Con ello, se pretende influir sobre el número de iteraciones que pueda tener el problema, para que este sea un número razonable.

El proceso por tanto será nuevamente, utilizar el *software* generador de problemas, para generar 100 problemas que tengan tres variables y dos restricciones, luego tres restricciones, después cuatro restricciones, y así sucesivamente hasta llegar a

seis restricciones. Nuevamente se ha llegado hasta seis restricciones porque el tiempo de ejecución empieza a ser alto, y serían problemas muy difíciles de resolver a mano, ya que aparecerían operaciones con matrices de 6x6.

Los resultados obtenidos al ejecutar *SimplexSolver*, son los siguientes:

	2 Restric.	3 Restric.	4 Restric.	5 Restric.	6 Restric.
Tiempo de ejecución de 100 problemas					
Tiempo de ejecución(ms)	3806	126051	1558630	3297039	15031413
Tiempo de ejecución(s)	3,806	126,051	1558,63	3297,039	15031,413
Tiempo de ejecución por problema(media)					
Tiempo de ejecución(ms)	38,06	1260,51	15586,3	32970,39	150314,1
Tiempo de ejecución(s)	0,03806	1,26051	15,5863	32,97039	150,3141

Tabla 72: Resultados del estudio en función de las restricciones (I)

En la siguiente gráfica se contemplan los datos anteriores:

Ilustración 34: Gráfico de tiempo de ejecución en función de restricciones (I)

Como se puede ver en la gráfica y en la tabla el tiempo de ejecución aumenta considerablemente cada vez que aumentamos una restricción. El motivo es que cada vez se realizan operaciones con matrices más complejas. El tiempo comienza a dispararse cuando se llega hasta las 5 restricciones. Hay que recordar que simplemente para hacer una iteración, habrá que invertir una matriz de 5x5, algo que sin duda es muy costoso de realizar a mano. Por lo tanto, se está corroborando que el *software* no solo da la posibilidad de resolver problemas que a mano, no serían muy costosos, sino que también permite resolver problemas que es impensable resolverlos a mano en un tiempo, que aunque obviamente mayor que en problemas con menos restricciones, es razonable.

Dentro de este apartado del estudio del rendimiento al aumentar el número de restricciones, se ha decidido incluir otra forma de estudio. Lo que se va a hacer en este caso es generar problemas, que únicamente tengan dos variables, y a su vez ir aumentando el número de restricciones. La idea ahora, es probar cuál es la eficiencia del *software* cuando se añade la búsqueda de la solución gráfica al mismo. Es decir, en este caso se va a ver cuál es la eficiencia del *software* en la obtención de la solución gráfica. Como se ha comentado antes, esta solución solo puede ser obtenida, con *SimplexSolver*, cuando el problema presenta sólo dos variables, de ahí que no se pueda hacer un estudio aumentando el número de variables y fijando el número de restricciones.

Los resultados obtenidos se muestran en la siguiente tabla:

	2 Restric.	3 Restric.	4 Restric.	5 Restric.	6 Restric.	7 Restric.
Tiempo de ejecución de 100 problemas						
Tiempo de ejecución(ms)	322810	518510	610256	1605559	873290	1515680
Tiempo de ejecución(s)	322,81	518,51	610,256	1605,559	873,29	1515,68
Tiempo de ejecución por problema(media)						
Tiempo de ejecución(ms)	3228,1	5185,1	6102,56	16055,59	8732,9	15156,8
Tiempo de ejecución(s)	3,2281	5,1851	6,10256	16,05559	8,7329	15,1568

Tabla 73: Resultados del estudio en función de las restricciones (II)

Como vemos los resultados anteriores son bastante dispares. Por ejemplo, se muestra un tiempo de ejecución menor con 6 y 7 restricciones que con 5. El motivo es que al generar problemas aleatoriamente, es complicado que se encuentren restricciones aleatorias que generen una región factible. Esta dificultad aumenta al aumentar el número de restricciones. La desventaja de que los problemas no tengan solución es que son resueltos mucho más rápidamente con el desarrollo que se ha realizado del método gráfico, por ello, los datos no se pueden considerar del todo reales.

Para contar con unos datos más realistas, se van a utilizar problemas específicos de los que se conoce que tienen solución, para así poder calcular de una manera más realista cuánto tarda en ejecutarse cada problema en función de las restricciones. Igualmente no será del todo determinante, puesto que el hecho de utilizar un único problema, puede dar lugar a casos extremos.

	2 Restric.	3 Restric.	4 Restric.	5 Restric.	6 Restric.	7 Restric.
Tiempo de ejecución(ms)	142	148	179	183	208	218
Tiempo de ejecución(s)	0,142	0,148	0,179	0,183	0,208	0,218

Tabla 74: Resultados del estudio con problemas preparados

En la siguiente gráfica se contemplan los datos anteriores:

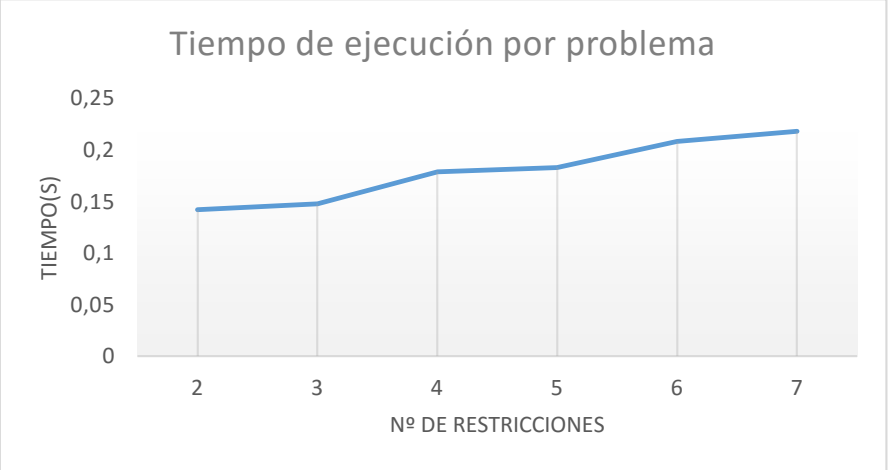

Tabla 75: Gráfico de tiempo de ejecución en función de restricciones (II)

Viendo la información recogida tanto al calcular los 100 problemas, como los problemas "preparados", se puede ver que el número de restricciones también afecta a la solución gráfica, aunque en este caso las diferencias son mucho más pequeñas. El motivo del estudio de este caso, es que el proceso de resolución es diferente que para la resolución mediante el método *Simplex*. En este caso se va a buscar las intersecciones entre las restricciones para delimitar la región factible. El motivo por tanto del aumento del tiempo de ejecución, viene dado porque al aumentar las restricciones, hay más puntos de intersección que tiene que comprobar el programa para delimitar la región factible. No es por tanto un problema a la hora de representar un mayor número de restricciones, sino de calcular puntos de intersección entre ellas y evaluarlos.

5.2.3. Conclusiones finales del estudio

Si se analizan las dos partes del estudio se puede concluir lo siguiente. Aunque es obvio que tanto el aumento de variables como el de restricciones influye en el aumento del tiempo de ejecución, ambos son incomparables en cuanto a los resultados obtenidos. La conclusión principal es que el número de restricciones marca mucho más el tiempo de ejecución.

El motivo de esta afirmación, es que pese a que el número de variables puede hacer que el problema sea más largo, puesto que puede haber más iteraciones, si fijamos el número de restricciones a un número no muy alto (como es el caso del presente estudio), simplemente tiene que hacer cálculos pero siempre de forma mecánica, ya que simplemente tiene que realizar los mismos cálculos pero muchas veces (tantas como iteraciones haya). Por otro lado, como ya se comentó más arriba, el número de restricciones hace que se trabaje con matrices de orden igual a este número. Esto sería un problema menor, si no se hubiera utilizado la clase *rational*, para implementar los números *racionales*.

El hecho de usar la clase *rational*, hace que se hayan tenido que reprogramar los métodos de operaciones con matrices (inversión, multiplicación…). Estos métodos

obviamente, son mucho menos eficientes, que los que pueda presentar la librería *numpy* para trabajar con números de tipo *double* o *int*.

Por supuesto toda la eficiencia, se ve perjudicada por la inclusión de esta clase. También se nota en el número de variables, pero como el estudio ha demostrado, no es tan influyente en el resultado final.

Otra de las conclusiones obtenidas, es que pese a que durante todo el documento se ha dicho que se había sacrificado la eficiencia del *software*, para darle exactitud, lo cierto es que los resultados del estudio muestran que la eficiencia sigue siendo bastante buena. Es obvio que problemas con 7 variables o con 6 restricciones, tienen un tiempo de ejecución que puede parecer elevado, pero si pensamos en el coste en tiempo que tendría resolver un problema de tal magnitud a mano, vemos que un tiempo de ejecución inferior a 3 minutos como presenta el sistema para estos problemas, es bastante razonable.

Por otro lado, habría que decir, que problemas de este tamaño no son demasiado habituales, sobre todo en el ámbito académico. Lo más común es que los problemas no tengan más de 3 ó 4 variables y no más de 3 ó 4 restricciones. Como demuestra el estudio el tiempo de ejecución para esos valores ronda los 15 segundos en el peor de los casos, por tanto, para los casos más comunes se puede afirmar que el tiempo de ejecución es pequeño teniendo en cuenta la exactitud que el resultado presenta.

Respecto a la parte del estudio que hace referencia a la solución gráfica, podemos sacar la siguiente conclusión. Aunque el número de restricciones afecta al tiempo de ejecución, no se produce un incremento tan grande de los tiempos como en el caso del método *Simplex*, cada vez que se añade una restricción. El motivo es que en este caso no tiene que hacer cálculos con matrices, sino simplemente tiene que calcular intersecciones entre restricciones, algo que sin duda, es computacionalmente menos costoso. Se podría afirmar por tanto, que si se trata de un problema de dos variables, y muchas restricciones, se podría optar por buscar la solución gráfica en lugar de buscar la solución mediante el método *Simplex*, ya que podría resultar mucho más rápido.

Todas las conclusiones del estudio, han sido sacadas sobre problemas generados aleatoriamente. El hecho de que sean aleatorios, hace imprevisible el resultado, y por ello se ha decidido en cada ejecución, generar 100 problemas, para reducir el impacto de posibles resultados extremistas. Además otro factor a tener en cuenta es el tamaño de los números que se utilizan. Para este estudio se ha decidido no usar valores demasiado grandes, ya que son los que normalmente presentan estos problemas en el ámbito académico.

El tamaño de los números puede ser determinante en el tiempo de ejecución. Esto se debe a que al incluir la clase *rational*, todo número es convertido a este tipo. Como se vio en el desarrollo, al crear un tipo de la clase *rational*, se calculan los divisores primos del numerador y el denominador, luego si tenemos valores muy elevados este cálculo se puede hacer demasiado largo. Esto es algo que afecta tanto a la representación gráfica como a la resolución por el método *Simplex*, pero por otro lado,

el tener guardados estos divisores agiliza las operaciones con números de la clase *rational*, ya que permite simplificarlos más fácilmente. En conclusión, aunque esta forma de implementación genere un mayor tiempo de ejecución al crear los tipos, es un tiempo que se gana al hacer las operaciones.

Capítulo 6
Conclusiones y líneas futuras

6. Conclusiones y líneas futuras

En los siguientes apartados se van a exponer las conclusiones alcanzadas una vez finalizado el proyecto y además, cuáles serían los pasos a seguir en un futuro, teniendo como base el proyecto actual.

6.1. Conclusiones

Una vez el proyecto ha terminado conviene saber si se han alcanzado los objetivos previstos en un primer momento. Para ello, se va a analizar, en qué medida el sistema es eficaz, eficiente y si se han cumplido los plazos previstos en cuánto al tiempo de desarrollo.

Un sistema será eficaz en la medida que produzca el efecto deseado. Se buscaba un sistema que fuera capaz de resolver problemas de programación lineal, de manera automática, ofreciendo además diferentes servicios, como guardar la solución o mostrar el desarrollo del problema. Pues bien, como se ha visto durante los casos de prueba del apartado 5.1, eso es algo que se ha conseguido.

En este caso, podemos hablar de eficiencia, como la velocidad con la que el sistema realizará su función. Como se ha comentado durante repetidas veces, la eficiencia no era un objetivo fundamental del proyecto, puesto que se da prioridad otros aspectos como la exactitud de la solución. Aun así, como se puede ver en el estudio realizado en el apartado 5.2, se obtienen unos tiempos, que se pueden considerar bastante eficientes.

En cuanto al tiempo empleado para el desarrollo, hay que decir que el desarrollo se ha terminado en el plazo esperado. Durante el apartado 7, se podrá comprobar que hubo que realizar algunos ajustes, durante la realización de algunas tareas, pero que no han impedido que el proyecto se finalice a tiempo.

En conclusión, se puede decir que se ha desarrollado un sistema es eficaz, porque resuelve la necesidad que se esperaba que resolviera. Se puede decir también, que se ha desarrollado un sistema eficiente, porque a pesar de sacrificar cierta eficiencia en favor de otras características, se alcanzan unos tiempos razonables, sobre todo con el tipo de problemas más habituales. Y finalmente, se puede decir que el desarrollo se ha terminado dentro del tiempo esperado.

Por último, habría que plantearse qué ofrece *SimplexSolver*, que lo diferencie de otros sistemas que se pueden encontrar en internet.

- La exactitud de su solución. No todos los sistemas ofrecen una solución exacta a los problemas, sino que únicamente muestran la solución en forma decimal.
- La posibilidad de obtener la solución gráfica. En comparación a otros sistemas, donde solo se puede obtener la solución mediante el método *Simplex*.
- Mostrar el desarrollo del problema. Contar con la posibilidad de ver todos y cada uno de los pasos, hasta alcanzar la solución final, es algo muy

importante sobre todo a nivel docente, cuando una persona está aprendiendo a resolver este tipo de problemas.

- Contar con una librería para usar todos los servicios. Ningún sistema, cuenta con la posibilidad de que puedas acceder al código, y usar su librería para desarrollos propios basados o no en el método *Simplex*, que sí ofrece *SimplexSolver*. Es decir, se da la posibilidad de usar la librería *Simplex.py*, para realizar desarrollos que requieran de ciertos servicios matemáticos que esta librería contiene.
- Un manual de usuario. *SimplexSolver* presenta un manual, con el que el usuario puede interactuar, haciendo sus propias pruebas y comprobando como funciona, tanto el sistema completo como los métodos de la librería.

Es decir, *SimplexSolver* reúne las características que presentan la mayoría de sistemas por separado, en uno sólo y además añade algunas mejoras más, como el comentado manual de usuario.

6.2. Líneas futuras

A continuación, se van a comentar las posibles mejoras que podrían introducirse en el proyecto, en un futuro.

6.2.1. Eficiencia

Como se ha comentado ya durante el presente documento, el programa *SimplexSolver*, no tiene la eficiencia como su cualidad más importante. El motivo, es que se ha dado menos importancia eficiencia en favor de la exactitud de la solución, es decir, se ha preferido que el *software* tarde más en resolver un problema, pero que lo haga de forma exacta.

Para alcanzar la exactitud del problema, se ha utilizado una clase que implementa los números *racionales*. El hecho de implementar esta clase hace que haya que implementar también todas las operaciones y estructuras necesarias para la resolución del método *Simplex*. Al usarse estas nuevas estructuras y operaciones, sin duda, el problema se ralentiza.

Pues bien, una posible mejora, sería buscar una implementación tanto de los números *racionales*, como de sus estructuras y operaciones, que permitan que el programa sea mucho más rápido.

6.2.2. Interfaz gráfica

SimplexSolver, recibe el problema a resolver en un archivo de texto plano. Esto hace que sea muy fácil para el usuario construir un problema y pasarlo al *software* para su solución. Por otra parte, esta forma de introducción es más susceptible de errores por parte del usuario, que podría no introducir el problema en el formato correcto.

Por lo tanto, sin perder de vista esta forma de introducir los problemas, puesto proporciona mucha flexibilidad al usuario, se podría desarrollar otra forma de introducción.

La idea sería crear una interfaz gráfica, en la que el usuario indique primero el número de variables y de restricciones, y después el resto de datos del problema. Se le proporcionarían también botones con todas las acciones posibles.

A pesar de que pudiera ser más tedioso para el usuario rellenar todos los campos, esta forma de usar el programa contaría con múltiples ventajas. Por un lado, el usuario no tendría que conocer los comandos de ejecución, sino simplemente pulsar un botón. Por otro lado, sería menos susceptible a fallos en el formato de introducción del problema, y además el usuario no tendría que consumir espacio en guardar archivos que contengan problemas.

6.2.3. Generación de solución en *Latex/*PDF

SimplexSolver ofrece al usuario la posibilidad de guardar la solución en un archivo. El archivo será simplemente un archivo de texto plano, donde el usuario podrá volver a consultar la solución, siempre que lo desee.

La idea de esta mejora, sería que la solución se generara en un formato *Latex*. Para ello, se tendría una plantilla estándar, donde simplemente habría que ir rellenando los valores obtenidos al resolver el problema.

Una vez obtenida la solución en *Latex*, se convertiría de este formato, a un documento PDF donde quedaría el problema perfectamente presentado.

Las ventajas de esta parte, serían que la solución quedaría presentada en un formato mucho más estético, además de quedar perfectamente presentada y organizada, acompañada de explicaciones que faciliten su comprensión. La principal desventaja, sería que el usuario debería conocer *Latex*, además de tener un conversor de *Latex* a PDF.

6.2.4. Creación de problemas

Esta mejora consistiría en ir un paso más allá, en los servicios que ofrece el sistema. La idea es que sea capaz de generar problemas de forma aleatoria, sujeto a ciertas condiciones introducidas por el usuario.

Por lo tanto, el usuario introduciría sus condiciones (por ejemplo, un problema con tres restricciones y dos variables, cuya solución sea no acotada) y el sistema generaría de forma aleatoria un problema. El problema además podría guardarse en un archivo de texto plano, y en el formato apropiado para que sirviera de entrada a *SimplexSolver*. De esta forma el usuario, podría crear tantos problemas como quisiera, y además ir viendo el desarrollo de su solución.

6.2.5. Representación en 3D

SimplexSolver ofrece la solución gráfica para un problema de programación lineal siempre que éste tenga únicamente dos variables de decisión. Como ya se ha comentado, a la hora de representar un problema cada variable genera una dimensión, luego como mucho se podría representar una solución de un problema que tuviera tres variables.

Hacer esto en un papel, proporciona una representación que no tiene demasiado valor, puesto que no se puede ver la región factible en su totalidad. Sin embargo, utilizar la representación en 3D en un ordenador, puede permitir al usuario moverse en la representación y ver la región factible completa.

Luego esta mejora consistiría en extender la representación gráfica, a aquellos problemas con tres variables de decisión, aplicando el método de resolución gráfica de la misma manera que se hacía con dos variables, teniendo en cuenta algunos detalles, como que ahora la región factible la formarán intersecciones de planos, y no de rectas como ocurría antes.

6.2.6. Versiones para móviles

Podría generarse una aplicación para móviles que realice los servicios que ofrece *SimplexSolver*, incluyendo también la resolución gráfica. Esto permitiría que un usuario pudiera resolver un problema en cualquier lugar, sin necesidad de tener que recurrir a un ordenador.

Se podría dar la posibilidad incluso, de que el usuario introdujera el problema escaneándolo mediante la cámara de su teléfono móvil, y que el sistema lo reconociera automáticamente, para pasar a continuación a resolverlo.

Habría que estudiar la viabilidad de esta mejora, puesto que la capacidad de cómputo de un dispositivo móvil, no es la misma que pueda tener un ordenador, luego podría ralentizar mucho el dispositivo, además de ser poco eficiente.

Otro inconveniente, sería que habría que utilizar un lenguaje de programación, que sea compatible con dispositivos móviles, y con los distintos sistemas operativos que estos presentan.

Capítulo 7
Planificación y presupuesto

7. Planificación y presupuesto

En este apartado se van analizar los recursos consumidos al realizar el proyecto.

7.1. Planificación de tareas

En este apartado, se van a mostrar las tareas en las que se ha dividido el desarrollo del proyecto. En primer lugar se mostrarán las tareas acompañados de una planificación que se hizo al comienzo del proyecto para a continuación, mostrar las tareas, con la duración real que han tenido. Ambos apartados irán complementados con un diagrama de Gantt, que permitirá visualizar la planificación de las tareas y cuáles de estas requieren que termine otra, o cuáles de estas se pueden solapar.

7.1.1. Estimación inicial

Se muestran aquí las tareas en las que se dividió el desarrollo, así como la duración estimada de cada una de ellas, que se realizó al comienzo del proyecto.

Tarea	Duración(Semanas)	Fecha inicio	Fecha fin
Aprendizaje de *Python*	7 semanas	5/10/2015	20/11/2015
Diseño de la aplicación	2 semanas	23/11/2015	04/12/2015
Primera versión de la resolución del	4 semanas	07/12/2015	01/01/2016
Mejoras y correcciones de la primera versión	2 semanas	05/01/2016	15/01/2016
Implementación versión final de la resolución del *Simplex*	2 semanas	18/01/2016	29/01/2016
Realización de manuales de usuario y programador	1 semana	01/02/2016	05/02/2016
Implementación Solución gráfica	3 semanas	08/02/2016	26/02/2016
Implementación con clase *rational*	1 semanas	29/02/2016	04/03/2016
Pruebas	1 semana	07/03/2016	11/03/2016
Documentación	15 semanas	01/02/2016	14/05/2016
Presentación	2 semanas	16/05/2016	27/05/2016
Seguimiento	33 semanas	5/10/2015	27/05/2016

Tabla 76: Estimación inicial

Hay que hacer algunas puntualizaciones a la tabla anterior. Para contabilizar las semanas no se han tenido en cuenta fines de semana, ya que durante estos días no se trabajó, luego las semanas se contabilizan de lunes a viernes. Respecto a la duración de la tarea de documentación, aunque se empieza a documentar desde el mismo momento que comienza el proyecto, se ha tenido en cuenta el comienzo previsto para empezar a hacer documentación de una forma más completa, que será cuando se tenga parte del desarrollo completo. Obviamente, antes de esto se realizarán tareas de documentación para planificación, análisis, diseño…

Por último, como se ve, la tarea de seguimiento abarca todo el proyecto, puesto que de principio a fin, se ha ido comprobando que el proyecto iba según lo previsto.

A continuación, se muestra un diagrama de Gantt, con las tareas, para ver mejor la planificación:

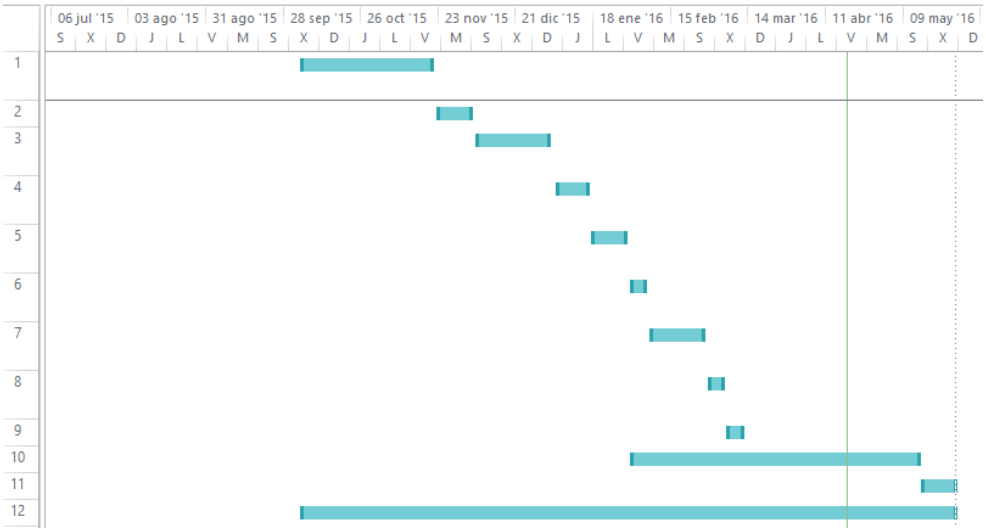

Ilustración 35: Diagrama Gantt Estimación Inicial

7.1.2. Estimación real

En la siguiente tabla (se tienen en cuenta las mismas consideraciones del apartado anterior) se muestra la duración real de las tareas, establecidas al principio del proyecto:

Tareas	Duración(Semanas)	Fecha inicio	Fecha fin
Aprendizaje de *Python*	6 semanas	5/10/2015	13/11/2015
Diseño de la aplicación	1 semanas	16/11/2015	20/11/2015
Primera versión de la resolución del *Simplex*	5 semanas	23/11/2015	25/12/2015
Mejoras y correcciones de la primera versión	2 semanas	28/12/2015	08/01/2016
Implementación versión final de la resolución del *Simplex*	3 semanas	11/01/2016	29/01/2016
Realización de manuales de usuario y programador	2 semanas	01/02/2016	12/02/2016
Implementación Solución gráfica	4 semanas	15/02/2016	11/03/2016
Implementación con clase *rational*	2 semanas	14/03/2016	25/03/2016
Pruebas	1 semana	28/03/2016	01/04/2016
Documentación	17 semanas	01/02/2016	27/05/2016
Presentación	2 semanas	30/05/2016	10/06/2016
Seguimiento	36 semanas	5/10/2015	10/06/2016

Tabla 77: Estimación real

Como se puede ver se han producido algunas diferencias en cuanto a lo previsto. Las tareas de aprendizaje del lenguaje *Python* y del diseño de la aplicación, se han terminado en una semana menos de lo previsto. Estas dos semanas de ganancia permiten compensar, el retraso en algunas de las demás tareas. Aun así, el proyecto en total, se ha retrasado tres semanas en cuanto a lo previsto en la planificación inicial. En el siguiente diagrama de Gantt, se puede ver mejor como queda la planificación final, ya con los tiempos reales:

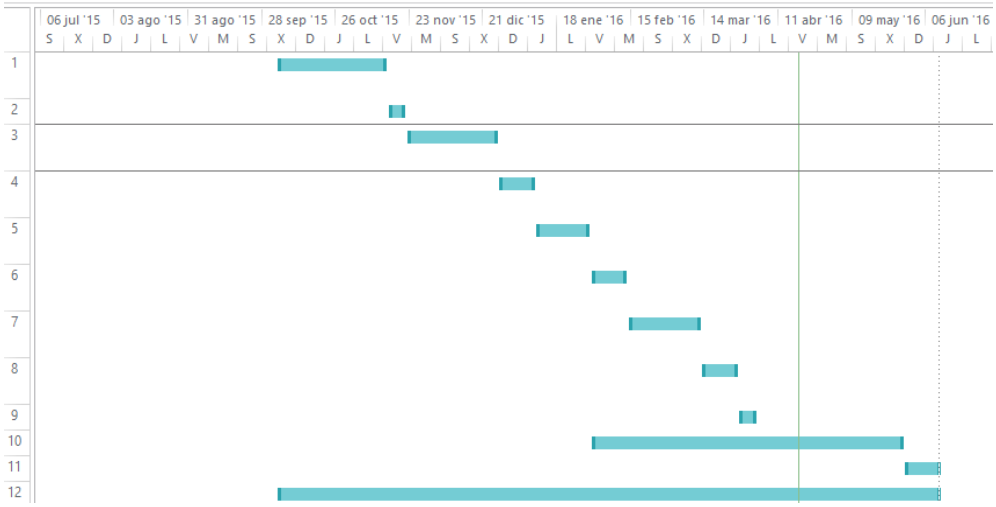

Ilustración 36: Diagrama Gantt Estimación Real

7.2. Planificación de los recursos

En este apartado, se van a mostrar los recursos utilizados para el desarrollo del proyecto. Se va a separar en hardware y *software*, para distinguir los elementos físicos, de aquellas herramientas *software* utilizadas.

7.2.1. Recursos hardware

- Sony Vaio, Intel® Core™ i5-3210M, 2.50 GHz, 6 GB RAM, Windows 64 bits

7.2.2. Recursos *software*

- Sistema operativo:
  - Windows 8
- Entornos de programación *Python*:
  - Komodo Edit 9
  - Ipython notebook
  - CMD Windows
- Lenguajes de programación:
  - Python 3.4
- Procesador de texto:
  - Microsoft Word 2010
- Diagramas UML:
  - Star UML
- Gráficos y otros diagramas:
  - Microsoft Excel 2010

- Presentación:
    - Microsoft Power Point 2010

7.3. Análisis económico

En este apartado, se va a realizar un análisis desde el punto de vista económico del proyecto desarrollado, teniendo en cuenta, en primer lugar los recursos utilizados, para después obtener un presupuesto final.

7.3.1. Recursos

En este apartado se va a analizar el coste de los recursos utilizados. Para ello, se va a hacer un desglose en tres tipos de recursos, a saber: recursos humanos, recursos hardware y recursos *software*.

- **Recursos humanos**: en el proyecto van a participar dos personas, una de ellas tendrá el rol de programador o desarrollador, mientras que la otra persona desempeñará funciones de supervisor del proyecto. Para calcular los costes de estos recursos, se realizará una estimación aproximada, en función del salario medio de este tipo de desempeños.
- **Recursos hardware**: estos recursos serán los comentados en el apartado 7.2.1. Para calcular su coste, se tendrá en cuenta el valor inicial dividido entre el período de vida en semanas estimado para el recurso, multiplicado por el número de semanas que ha durado el proyecto.
- **Recursos *software***: estos recursos serán los expuestos en el punto 7.3.2. Se tendrá en cuenta el coste de adquisición de la licencia de los mismos. Solo se tendrán en cuenta aquellos que no sean *software* libre y requieran de una licencia de adquisición.

7.3.2. Presupuesto

Para calcular el presupuesto se han tenido en cuenta las 36 semanas que ha durado el proyecto. De estas, como se indica en la planificación, hay que descartar los fines de semana. Además, para el rol de programador habría que contabilizar que dedica una media de 20 horas semanales, durante todo el proyecto. Esto haría un total de 720 horas de duración del proyecto por parte del programador.

Por parte del supervisor, emplea una media de 2 horas semanales para la corrección y seguimiento del proyecto, luego sus horas ascienden a 72 horas durante todo el proyecto. En la siguiente tabla se muestra el coste de los recursos humanos:

Recurso	Horas empleadas	Coste €/Hora	Coste Total
Programador	720 horas	20 €/Hora	14.400 €
Supervisor	72 horas	25 €/Hora	1.800 €
TOTAL			**16.200 €**

Tabla 78: Presupuesto recursos humanos

En las siguientes tablas, se muestran los costes de los recursos hardware y *software* empleados durante el proyecto. Para calcular las horas del tiempo de vida, se estimará un uso medio de 8 horas diarias.

Recurso	Inversión	Tiempo de vida	Tiempo de uso	Coste Total
Sony Vaio	950 €	14400 horas	720 horas	47.5 €
TOTAL				**47.5 €**

Tabla 79: Presupuesto recursos hardware

Recurso	Coste	Tiempo de vida	Tiempo de uso	Coste Total
Windows 8	95 €	8640 horas	720 horas	7.9 €
Microsoft Office 2010	99 €	2880 horas	340 horas	11.6 €
Star UML	70 €	2880 horas	5 horas	0.12 €
Komodo edit	295 €	2880 horas	520 horas	53.2 €
			TOTAL	**72. 82 €**

Tabla 80: Presupuesto recursos software

Si se suman todos los costes humanos, de *software* y de *hardware*, obtendremos el coste total del proyecto, como se puede ver en la siguiente tabla.

Recurso	Coste
Recursos Humanos	16.200 €
Recursos Hardware	47.5 €
Recursos *Software*	72. 82 €
TOTAL	**16320.32 €**

Tabla 81: Presupuesto total

El coste total del proyecto, ascenderá por tanto, a 16320.32 €.

Capítulo 8
Referencias

8. Referencias

[1] *Programación lineal.*
https://es.wikipedia.org/wiki/Programaci%C3%B3n_lineal

[2] *Historia de la programación lineal*.
http://www.sectormatematica.cl/contenidos/prolinhis.htm

[3] *Los números racionales.*
https://es.wikipedia.org/wiki/N%C3%BAmero_racional

[4] *Los números irracionales.*
https://es.wikipedia.org/wiki/N%C3%BAmero_irracional

Capítulo 9

English Summary

9. English summary

This chapter is an English summary of the project.

9.1. Introduction

Linear programming is an optimization maths field, in which an optimization of a linear function is looked for, in a way that the variables of this function must satisfy a set of restriction that are also linear.

The linear programming problems can be solved by several methods, however this project will focused on two of them: *Simplex method* and *Graphical method*.

Both have some problems. *Simplex* needs that many maths operations are solved and it is time-consuming and tedious. *Graphical method* is faster and simpler than *Simplex*, but it is difficult to use it by hand, because drawing the solution in a paper is not as exact as doing it with a computer. This final degree project pretends to get over these problems or, at least, try to minimize them.

The project will be developed in order to help people who are learning and studying linear programming and need anything to practise, notice their mistakes, etc.

In conclusion, the project has two main purposes. On the one hand, it is to have a software that can help students to practise and improve their skills with *Simplex method* and *Graphical method*.

On the other hand, it is to have a library with maths services (including services to solve linear programming problems) that can be used in his own develop.

9.2. Simplex method

Simplex method is based on a set of iterative steps that are applied to a linear programming problem and let us get a solution of it. If a problem doesn`t have solution, the method shows us as well. Now, steps of the *Simplex method* are going to be explained.

9.2.1. Simplex input

At first, *Simplex method* receives a problem which has a linear function and some restrictions about the variables of the function. The form of the problem is:

$$Optimize\ z = c^T x$$
$$subject\ to\ \ Ax \leq b\ , x \geq 0$$

z is the function which has to be optimized; c^T is the vector of the function; A a is the coefficients matrix which contains all coefficients of the restrictions and b is the resources vector.

9.2.2. Normal standard form

Simplex method can be only used when the problem is in a distinct form which is called normal standard form. A problem is in normal standard form when:

- The type of the function is maximization.
- The sign of all restrictions is =.

- All decision variables are ≥ 0.
- All resources are not negative, in other words, no element of the resources vector can be negative.

Obviously, not all problems are in standard normal form, but it doesn`t mean that *Simplex* can`t be applied. There are some transformations to put a problem in standard normal form.

When the problem is in standard normal form, *Simplex* can start.

9.2.3. Base's selection

Firstly, a base must be selected. The base will be a square matrix, composed of variables' coefficients of the restrictions. They are the columns of A matrix. In the first iteration, the base must be an identity matrix, so if there is no columns in A matrix to make an identity matrix, some transformation must be applied to the restrictions.

9.2.4. Solution of the iteration

Then the solution of the iteration is calculated. It is got by multiplying inverted base matrix and resources vector.

9.2.5. Function iteration value

Next, with the solution of the iteration, function value is calculated. It is necessary to multiply each variable value and the coefficient of the variables in the function for this iteration. This is, for example, if the function is $z = 3x_1 + 4x_2 + 6x_3 + 9x_4$ and the variables of the iteration are x_1 and x_2, the vector of the function for this iteration will be (3 4).

9.2.6. Input rule

In this part of the iteration, it is calculated the variable which will enter in the next iteration. The formula to obtain it is:

$$z_j - c_j = c_{B_0}^T B_0^{-1} a_j - c_j$$

This formula is calculated for every variable which is not in the iteration. $c_{B_0}^T$ *is* the vector of the function for this iteration, B_0^{-1} is the base of the iteration inverted, a_j is the column of A matrix for that variable and c_j is the coefficient in the function for that variable.

Variable which has the most negative value will enter in the next iteration.

9.2.7. Output rule

Finally, the last step of the iteration is to calculate the variable which will leave the iteration. The formula is:

$$O = min_{i \in B}\{\frac{x_i}{y_{ii'}}\}$$

In this formula x_i is the solution of the iteration and $y_{ii'}$ is the y value for the variable which will enter in the next iteration. y is calculated by $y_j = B_0^{-1} a_j$.

In this operation, values that have negative or zero denominator will be discarded. Then minimum value gives us the variable that leaves.

9.2.8. Solution improvement

When the iteration is finished, the next iteration starts. Firstly, the new base is calculated. The base of the next iteration will be the previous iteration base, but the variable calculated in the input rule is added, and the variable resulting of the output rule is eliminated. Then the steps are the same that in the previous iteration.

9.2.9. End of the problem

Therefore, the way to solve a problem is to develop iterations of *Simplex,* but, when is the problem finished? There are several possibilities:

- When input rule doesn`t have any negative value. In this case, the problem is finished, the solution is unique and it will be the solution of the iteration.
- When input rule has a zero value and the other values are positives. The problem is finished and it is said that the problem has an infinite number of solutions.
- When it can't go out any variable in the output rule. If every value of the output rule has a negative or zero value, it can`t leave any variable. The problem is over and it is said that the problem is not bounded because function value can be improved indefinitely.
- When an artificial variable takes a positive value. Artificial variables are added to the function with a - ∞ coefficient, so if the value of the variable is positive, the function will get a maximum value of - ∞, and it doesn't make sense. In this case, it is said that there is no solution for the problem or that the problem is not feasible.

9.2.10. Duality

When a linear programming problem is solved, it is called the primal problem. Well, there is another problem, which can be obtained from the primal problem, it is called dual problem. This problem gives us some information about the primal problem because they have some properties in common.

In order to obtain the dual problem, primal problem must be in a form called symmetrical form of maximization. A problem will be in symmetrical form of maximization whether:

- The kind of the function is maximization.
- The signs of all restrictions are $\leq$.
- All variables have positive or zero values.

Some transformations could be necessary to put a problem in this form.

Once the problem is in symmetrical form of maximization, the dual problem is obtained from the primal problem by this way:

Primal problem		Dual problem
$\max Z = c^T x$		$\min w = b^T x'$
$Ax \leq b$	⇨	$A^T x' \leq c$
$x \geq 0$		$x' \geq 0$

9.2.11. Solve dual problem

Dual problem is solved by multiplying the function vector of the iteration (that is to say, this vector only has the coefficients of the variables that belong to the iteration) and the inverted base of the last iteration of the primal problem.

The solution of the dual problems gives us the contribution for resource unit. Contribution for resource unit is how a resource increases or decreases the function. For instance, if the dual problem solution (supposing that the last iteration of the primal problem had the variables x_1 and x_2) is (2 1), this means that for each unit that a resource of x_1 increases, the function increases in two units; and for each unit of x_2, the function increases one unit.

9.3. Rational numbers

Rational numbers are those that can be expressed as the quotient between two whole numbers that must be different of 0.

The use of this kind of number to solve a problem with the *Simplex method* is so convenient. The main reason to use this kind of numbers is that it is the only way to get the solution of the problem with accuracy.

With the following example, the accuracy of the operation with rational numbers is demonstrated:

Operation with rational numbers	Operation with real numbers
$\frac{1}{7}$ x $7 = 1$	1/7=0,1428571…
	0,1428571 x 7 = 0,9999997

9.4. Graphical method

Graphical method is an alternative to the *Simplex* in order to solve a linear programming problem. This method can be only used when the problem has two or three variables. The reason is that each variable generates a dimension in the plane, therefore, with more than three variables, the problem can`t be drawn.

Problems with three variables can be solved with this method, however when it is done in a paper by hand, it is quite difficult and the final representation doesn`t show us the whole solution.

The steps to solve a problem with the *Graphical method* are the following:

1. A system of Cartesian axis is drawing with as dimensions as variables the problem has.
2. Each restriction is drawn as a region. With two variables, each restriction is represented with a line, and with three variables, each restriction is represented as a plane.
3. Intersection between restrictions is calculated. In problems with two variables, the intersection between two restrictions will be a point and, with three variables, the intersection will be a line. Composition of the intersections between restrictions will be called feasible region.
4. The function is checked in all extreme points of the feasible region and it is chosen the values that optimize the function. It is important that we are only interested in the solutions of the first quadrant, because all variables must have a positive or zero value.

The same cases that we obtained with the *Simplex method* can be obtained with this method:

- **Unique solution**: The problem has unique solution if there is only one point of the feasible region which optimizes the function.
- **Infinite solutions**: The problem has infinite solutions if two points give the same function value. The solution of the problem will be the line that puts together these two points. If the problem has three variables, the solution will be the plane formed by the two lines which let us obtain the same value of the function.
- **No solution:** The problem won`t have solution if there is no feasible region formed by the restrictions of the problem.

9.5. Objectives

The services that the system should provide are the following:

- The system will be able to solve a linear programming problem using *Simplex method*.
- The system will be able to solve a linear programming problem using *Graphical method* if the problem has two variables.
- The system will be able to show the solution of a problem. The user will chose if he wants to see only the solution or every step of *Simplex* as well.
- The solution will be showed in console or written in a file if user indicates it.
- The system will have a library that besides required methods for *Simplex method*, it will have methods for doing other maths operations.
- The system will be able to generate dual problem of a problem introduced by user and solve it.

9.6. Development

The final version of the system will have several parts. Firstly, it will have a part where rational numbers will be implemented. Secondly, there will be a part which will contain every maths service the project has (it will be a library). A section of these maths services will be those that let us apply *Simplex* and *Graphical method*. Finally, the system will have a part that will solve linear programming problems automatically. This final part, will use the library that contains all maths services.

Once the software has been developed, three classes composes the whole system. These classes are described below.

9.6.1. Class Rational.py

This class implements rational numbers. Each object of this class has a numerator and a denominator. When a natural number is represented, it takes 1 as denominator. For each rational number, prime dividers of numerator and denominator are calculated and are stored in the object. In this class, there are also methods to overwrite all operations that can be done with real numbers in order to use rational numbers instead.

9.7.2. Class Simplex.py

This class is the central part of the system. It contains all method to apply *Simplex* and *Graphical method*. The class uses rational numbers in all of its operations so it imports Rational.py. It can be said that this class is a library and it is divided in four parts:

- **Rational operations:** the methods of this section of the library implements some rational numbers operations. Most of them are used in other methods to check if parameters are correctly introduced.
- **Matrixes operations**: the methods of this part of the library implements some matrixes operations like inversion or multiplication. These operations are also implemented by a *Python* library called *Numpy*. The problem is that the methods of this library can only be used with normal types (*double*, *int*…) not with the system implementation of rational numbers, so it is necessary to redefine the operations in order to use them with rational objects.
- ***Simplex*:** the methods of this part let us develop every step of *Simplex method*. There are methods to solve an iteration, the input rule, the output rule, etc.
- **Graphical Solution:** the methods of this part let us calculate graphical solution of a problem. There are methods to calculate the feasible region, the optimal points of the feasible region, etc.

This library can also be used for any user in his own development, only by importing it. As we can see, the library contains method that implements maths operations (like inversion of an objects' matrix) so these can be useful for anyone who is developing a software that requires this kind of service. The only restriction is that the user must use *Python* in his development.

9.7.3. Class SimplexSolver.py

This class receives a linear programming problem and solves it. There is no methods in this class, it only calls to *Simplex.py* library and uses some of its method to solve the problem.

The problem must be passed in a plain text file, with the proper format. The format of the problem must be, for example:

min -1 -2

1 1 = 6

3 1 = 12

1 1 <= 16

The class receives some parameters and acts accordingly. The parameters that users can pass to the class can be:

- The input file with the problem.
- If the user wants to see all steps of the problem.
- The name of an output file where solution is stored in.
- If the user wants the solution of the dual problem.
- If the user wants the graphical solution of the problem (it is only available for problems with two variables).

According to these parameters, the class calls the library´s methods as appropriate.

9.7.4. Manuals

In order to complete the software, two manuals have been implemented with a tool called *ipython notebook*. *Ipyhon notebook* is a web tool that allows besides introducing text and images, executing *Python* code. In other words, the user can read explanations of every part of the software and can also see examples and make changes in them.

One of the manual is considered a user manual. It shows how to execute *SimplexSolver.py* in several different ways, explaining all of them. The user can see the result of each kind of execution.

The other manual is considered a developer manual. It shows how to execute each method of *Simplex.py* library and the results of it. With these examples, a user can know how to use the methods in his own development.

9.8. Results

An example of an execution of *SimplexSolver* could be the following (*archivo1.txt* is the file which contains the problem):

```
C:\workspacePython>C:\python34\python SimplexSolver.py --input archivo12.txt --graphic --dual
SOLUTION: x* =[3 , 1 , 4 , 0 , 0] ,  z* = 3 .There is a unique solution.
DUAL SOLUTION: x* =[0 , 1/3 , 5/3 , 0 , 0]
```

Ilustración 37: Example of execution. Simplex's solution.

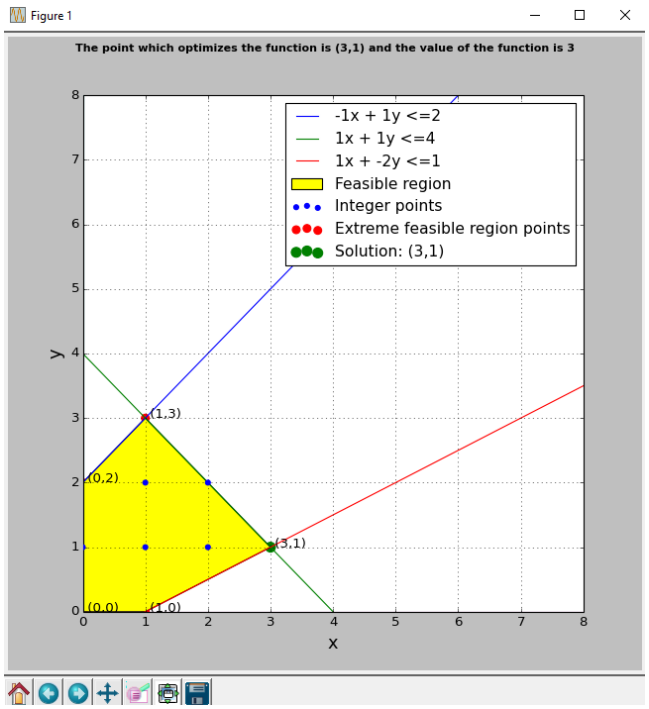

Ilustración 38: Example of execution. Graphical solution.

As it can be seen, the software shows the solution of primal and dual problem, an explanation of the solution and the graphical solution.

As it is said before, it is necessary to introduce rational numbers to obtain an exact solution. Rational numbers are not implemented in *Python* so a class that implements this numbers is created. This class is *Rational.py*. Introducing this class, the required accuracy is obtained, but obviously, the efficiency of the software decreases.

The main reason for this decrease is that methods, which implements operation between numbers, have to be reprogrammed. The new versions of these methods are not as efficient as *Numpy* (a *Python's* library for a wide set of maths operations) implementation of them. Another factor that should be taken into account is that for each rational number defined, it is necessary to calculate prime dividers of its numerator and its denominator, and it takes time, especially when the number is high.

Taking into consideration all this, in order to check the real efficiency of the system, a study has been done. The study is divided in two parts.

The first part of the study is focused on how affects the number of variables to the execution time of the software. In other words, the slower in solving a problem the system is, depending on the number of variables. In this part of the study, the number of restrictions are fixed to three restrictions. Firstly, problems with two variables and three restrictions are created, then with three variables, then with four variables…

This part finishes with problems that have seven variables and three restrictions. For each kind of problems, are created 100 of them and they are solved by *Simplex method*.

It can be observed that the time in solving a problem increases when every variable is added. This is because the probability of having more iterations in the problem increases with the number of variables, so the more variables the problem has, probably, the longer the problem is.

The second part of study, is the opposite of the first part. In this case, the number of variables is fixed to three variables and the number of restrictions are increased. This shows us how affects the number of restrictions to the execution time of the software. As in the first part of the study, one-hundred problems for each kind are created and solved *Simplex method*, so firstly, problems with three variables and two restrictions are generated, then with three restrictions, then with four restrictions…

This part finishes with problems that have three variables and six restrictions. At this point, it is observed that if the number of the restrictions grows, the time in solving the problem grows too. The reason is that the number of restrictions determinates the size of the matrixes that *Simplex method* require. With six restrictions, 6x6 square matrix are required and operations with matrixes of this size are really hard and long. It is almost unthinkable to try to solve problems of this shape by hand, because the time in inverting a 6x6 matrix, for example, could be too long.

If the two parts of the study are compared, a conclusion is obtained. The number of restrictions is largely more influential than the number of variables. On the whole, it can be said that the fact of having to operate with big matrixes, takes more time than a longer problem in iterations but with smaller matrixes.

The second part of the study has been repeated, but using *Graphical method* in this case. Obviously, the number of variables is fixed to two now, because are the problems that the software can only solve using *Graphical method*. The other difference is that this time, the study finishes with problems that have two variables and seven restrictions instead of six restrictions.

The results are that the time grows also in *Graphical method* when restrictions increase but not as much as with *Simplex*. In this case, the time grows not because it has to do more operations, but because it has to calculate more intersection points between restrictions. Despite this, it is less costly in terms of efficiency to calculate intersections between restrictions than to do matrixes operations. In fact, if the problem only has two variables could be faster to use *Graphical method* than *Simplex* in order to solve it.

Before, it was said that using rational numbers made less efficient the software but more exact. It is true, but the study shows that the software continues being efficient. With big problems, which have seven variables or six restrictions, *SimplexSolver* takes about five minutes in solving them. This time is reasonable considering that to solve this kind of problem by hand would take hours.

Besides, problems of these sizes are not so common, especially in an academic atmosphere. People, and especially students, hardy ever work with problems which have more than three or four variables and three or four restrictions. The time in solving this kind of problems with *SimplexSolver* is about fifteen seconds, in the worst case. In conclusion, for the most common kind of problems, *SimplexSolver* is quite efficient.

Another factor that can be influential in the time of execution is the size of the numbers. If the numbers are too big, the time in calculating their prime dividers could be quite high, and an apparent short problem can turn out long.

9.9. Conclusion

Once the project is finished, it is important to know if objectives have been achieved. The project is going to be evaluated in three facets: efficiency, effectiveness and time.

The purpose of the project was to develop a software that could solve linear programming problems by using *Simplex* and *Graphical method*. As it can see in the results, *SimplexSolver* allows solving a problem with these two methods, so it can be said that the system is effective.

As the study shows, the system can be considered efficient, because common problems are solved in a short period of time and problems with many variables and restrictions, are solved in a reasonable time (less than five minutes).

In terms of time, the project has been finished in time. It was necessary to change some parts of the schedule, but these changes were made up for with other tasks that finished before the time expected.

Finally, it would be positive to know what features offer *SimplexSolver* that other systems don't.

- **An exact solution**. Some system allows using *Simplex*, even *Graphical method*, but most of them uses real numbers instead of rational, therefore, they don`t offer an exact solution as *SimplexSolver* does.
- **Graphical solution.** Not all system let us use *Graphical method*. The majority of them only let us use *Simplex method*.
- **Shows every step of *Simplex*.** The majority of the systems that can be found on internet don't allow seeing all steps of the *Simplex* application in a problem.
- **Library**. There isn't any system which allows access to its code and using it for whatever you want. Besides using *SimplexSolver*, every user can import *Simplex.py* and use its method in his own development.
- **Manuals**. There isn`t any system which has a use manual with the level of details and the possibility of interact with it as *SimplexSolver* has.

All in all, *SimplexSolver* has all features that other systems present, and also others that no system has, as the mentioned manuals.

9.10. Future steps

The improvements that system could have in the future are the following:

- Efficiency. Efficiency is not the best feature of *SimplexSolver*. The need of using rational numbers and its implementation make the system less fast. An improvement could be to do another implementation of rational numbers that let the system be faster, or at least, as fast as using real numbers.

- Graphical interface. *SimplexSolver* receives the problem in a plane text file. The problem must be in the proper format. It could be an improvement to have an interface in where user could indicate the problem by filling different fields. With this interface, user doesn`t have to be worried about the format of the problem and about the execution commands. The disadvantage is that to fill many fields could be tedious and tiring, so the current form of introducing a problem would be maintained, and user would decide which of them to use.

- Latex. If the user wants solution can be stored in a file, and user can check it whenever he wants. An improvement could be generate this file with the solution in Latex.

  Then, Latex can be converted to PDF, the solution would be presented in a more pretty and organized format. Explanations about the solution could be added in these PDF. The only disadvantage is that user would have to know Latex and also he would need a Latex to PDF converter.

- Generate problems. This improvement would consist in adding another functionality to the system. The idea is that *SimplexSolver* could generate problems randomly. The user would indicate the condition of the problem (the number of variables, the number of restrictions, the kind of solution…) and *SimplexSolver* would generate a random problem that satisfies these conditions.

- 3D representation. Graphical solution is only available when the problem has two variables. This improvement would consist in extending this availability to problems with three variables. In other words, *SimplexSolver* could make three dimension plots that would show the graphical solution of a three variables' problem.

- Mobile app. It would be great, if all of *SimplexSolver* services could be implemented in a mobile app. With this app, users could solve a problem everywhere. It would be necessary to check if mobiles are so powerful that they can support as operations as *SimplexSolver* does. Another point is that

the app would have to be developed in a mobile compatible language programming.